프로그람개발문고

Qt일반지식

새세대

교육위원회 교육정보쎈터
주체99(2010)

차 례

머리말 4
제 1 장. Qt 에 대한 소개 5
제 1 절. Qt 상업판 5
제 2 절. Qt 공개원천판 6
제 3 절. Qt 의 간단한 력사 7
제 2 장. Qt 의 시작 9
제 1 절. Qt 학습방법 9
제 2 절. 포쏘기유희의 작성 9
제 3 절. 도표의 작성 52
제 4 절. 간단한 응용프로그람 85
제 3 장. Qt 객체모형 95
제 1 절. Qt 객체모형 95
제 2 절. 객체나무와 객체소유자 95
제 3 절. 신호와 처리부 96
제 4 절. 메타객체체계 101
제 5 절. 속성 102
제 6 절. Qt 는 왜 신호와 처리부에 형판을 사용하지 않는가? 105
제 4 장. Qt 의 리용방법 107
제 1 절. 설치 107
제 2 절. Qt 플라그인 109
제 3 절. QDataStream 연산자들의 형식 112
제 4 절. 오유수정 115
제 5 절. 끌어다놓기 117
제 6 절. 사건과 사건려과기 122
제 7 절. 건반초점 124
제 8 절. 표준지름건 126
제 9 절. Qt Netscape 플라그인확장 127
제 10 절. ActiveQt 틀거리 128

제 11 절. Qt Motif 확장 130
제 12 절. Qt OpenGL 확장에서 X11 오버레이를 사용하는 방법 149
제 13 절. 응용프로그램그림기호의 설정 150
제 14 절. 쎄숀관리 152
제 15 절. 공유클라스 154
제 16 절. 형식개괄 157
제 17 절. Qt 형판서고 160
제 18 절. Qt 에서 스레드기능 164
제 19 절. 시계 168
제 20 절. Qt 창문부품들의 화상 170
제 21 절. Qt 응용프로그람의 전개 175
제 5 장. 기하학적배치 177
제 1 절. 배치관리자클라스 177
제 2 절. 자체의 배치관리기작성 181
제 3 절. 자리표계 184
제 4 절. 창문기하 187
제 6 장. 모듈 189
제 1 절. 캔버스모듈 189
제 2 절. IconView 모듈 190
제 3 절. 망모듈 190
제 4 절. Qt OpenGL 3 차원도형처리 197
제 5 절. SQL 모듈 198
제 6 절. 표모듈 231
제 7 절. 작업공간모듈 232
제 8 절. XML 모듈 232
제 7 장. 국제화 248
제 1 절. Qt 에 의한 국제화 248
제 2 절. 유니코드 255
제 8 장. 이식과 가동환경 257
제 1 절. 창문체계에 고유한 알아두기 257

제 2 절. Qt/Mac 문제 259
제 9 장. Qt/Embedded 263
제 1 절. Qt/Embedded 의 설치 263
제 2 절. 특성정의파일 264
제 3 절. Qt/Embedded 에 고유한 클라스들 268
제 4 절. Qt/Embedded 에 가속기그라픽스구동프로그람추가 272
제 5 절. Linux 틀완충기의 허용 273
제 6 절. Qt/Embedded 응용프로그람의 실행 277
제 7 절. Qt/Embedded 에서 문자입력 278
제 8 절. Qt/Embedded 의 서체 279
제 9 절. Qt/Embedded 위치지정장치조종 282
제 10 절. Qt/Embedded 환경변수 282
제 11 절. 자기의 응용프로그람을 Qt/Embedded 에 이식 284
제 12 절. Qt/Embedded 를 이식할 때 알아야 할 문제 284
제 13 절. Qt/Embedded 성능조정 285
제 14 절. VNC 봉사기로서의 Qt/Embedded 286
부록 1. Qt 클라스서고 287
부록 2. 수 2-9999 의 씨수표 308
참고문헌 310

머 리 말

위대한 령도자 **김정일**동지께서는 다음과 같이 지적하시였다.

《프로그람을 개발하는데서 기본은 우리 식의 프로그람을 개발하는것입니다. 우리는 우리 식의 프로그람을 개발하는 방향으로 나가야 합니다.》 (《**김정일**선집》 제15권, 196페지)

위대한 령도자 **김정일**동지의 현명한 령도에 의하여 오늘 우리 나라에서는 프로그람기술이 빠른 속도로 발전하고있다. 우리의 과학자, 연구사들은 우리 식 조작체계 《붉은별》을 개발하였으며 각종 도구들과 응용프로그람들을 개발하기 위한 연구사업을 활발히 진행하고있다.

우리는 정보공학을 전공하는 교원, 연구사들과 대학생들이 프로그람개발도구인 Qt에 의하여 프로그람을 능숙하게 작성할수 있도록 하기 위하여 《Qt일반지식》과 《Qt프로그람개발법》, 《Qt프로그람개발도구》, 《Qt실례프로그람》을 출판한다.

《Qt일반지식》에서는 Qt의 객체모형, 신호-처리부, 사건과 사건려파기를 비롯한 Qt프로그람이 기초하고있는 일반적인 지식을 서술한다.

《Qt프로그람개발법》에서는 Qt에 의한 GUI프로그람작성방법을 서술한다.

《Qt프로그람개발도구》에서는 Qt Designer를 비롯한 Qt에 부속되여있는 각종 도구들의 사용법을 서술한다.

《Qt실례프로그람》에서는 Qt로 작성한 실례프로그람들을 서술한다.

이 책들은 독자들이 표준 C++언어에 대한 지식을 소유하고있는것을 전제로 한다.

우리는 Qt의 일반지식과 프로그람작성법을 습득하고 우리 식의 조작체계에서 실행할수 있는 프로그람들을 더 많이 개발함으로써 나라의 프로그람기술을 한계단 더 발전시키고 인민경제의 정보화를 실현하는데 적극 이바지하여야 한다.

제1장. Qt에 대한 소개

Qt는 Trolltech에 의하여 개발관리되는 다중가동환경 C++ GUI응용프로그람개발도구이다. Qt는 예술적조화를 갖춘 GUI작성에 필요한 모든 기능을 프로그람개발자들에게 제공한다. Qt는 완전히 객체지향적이고 확장하기 쉽고 진실한 콤포넨트프로그람작성을 가능하게 한다.

Qt는 1996년초에 상업적인 소개가 있은 다음에 세계적범위에서 수천개의 성공적인 응용프로그람들의 기초를 형성하였다. 또한 Qt는 대중적인 KDE Linux탁상환경의 기초이며 Linux의 모든 주요배포물들의 표준구성요소이다.

Qt는 다음과 같은 가동환경에서 동작한다.

- MS/Windows – 95, 98, NT 4.0, ME, 2000과 XP
- Unix/X11 – Linux, Sun Solaris, HP-UX, Compaq Tru64 UNIX, IBM AIX, SGI IRIX 등
- Macintosh – Mac OS X
- 매몰형 – 틀완충기가 유지되여있는 Linux가동환경.

Qt는 여러가지 판으로 출하된다.

- Qt Enterprise판과 Qt Professional판은 상업용쏘프트웨어개발용으로 제공된다. 이것들은 상업용쏘프트웨어배포물이며 무료로 갱신 및 기술지원봉사를 포함한다. Enterprise판은 Professional판에 비하여 추가적인 모듈들을 제공한다.
- Qt공개원천판은 Unix/X11, Macintosh와 매몰형Linux용이다. 공개원천판은 오직 무료 및 공개원천쏘프트웨어개발용이다. 이것은 QPL(Q Public License)과 GPL(GNU General Public License)에서 무료로 구입할수 있다.

제1절. Qt상업판

Qt Professional과 Enterprise판은 Qt의 상업판이다.

Professional판이나 Enterprise판을 구입할수 있다면 상업권과 소유권, 비공개쏘프트웨어를 작성할수 있다. (인정된 공개원천허가를 리용하여 출하하기 위한 무료 및 공개원천쏘프트웨어를 개발하려면 Qt공개원천판을 사용할수 있다.) Qt상업판의 구입은 기술적지원과 갱신을 포함한다. Microsoft Windows용 Qt는 Professional과 Enterprise판에서만 사용할수 있다.

표 1-1. Professional과 Enterprise의 비교

Professional/Enterprise비교표	Professional	Enterprise
Qt기초모듈(도구, 핵심, 창문부품, 대화칸) 가동환경에 의존하지 않는 Qt GUI도구와 편의클라스들	X	X
Qt Designer Visual Qt GUI구축프로그람	X	X
그림기호보기모듈 사용자교제기능이 있는 픽스매프모임의 시각화	X	X
작업공간모듈 다중문서대면부유지	X	X
OpenGL 3차원도형처리모듈 Qt와 OpenGL의 통합		X

Professional/Enterprise비교표	Professional	Enterprise
망모듈 소케트와 TCP, FTP, 비동기DNS조사용의 가동환경에 의존하지 않는 클라스들		X
캔버스모듈 시각화, 도표 등을 위한 최적화된 2차원도형처리령역		X
표모듈 유연하고 편집가능한 표/표계산프로그람		X
XML모듈 SAX대면부와 DOM준위1을 갖춘 잘 형성된 XML 구문해석기		X
ActiveQt확장 Windows에서 ActiveX조종요소구축 및 주콤퓨터화를 지원		X
QMotif확장 Motif 호상존재와 이식의 지원		X

제2절. Qt공개원천판

공개원천쏘프트웨어(일명 무료쏘프트웨어)는 사용자들에게 일정한 권한을 주는 사용허가를 가진 쏘프트웨어이다. 특히 쏘프트웨어를 사용할 권한, 원천을 수정할 권한, 원천을 얻을 권한, 그것을 넘길 권한(같은 조항하에서)을 준다.

Trolltech는 Qt공개원천판을 제공하여 공개원천쏘프트웨어개념을 지원한다. Qt의 이 판을 사용하여 공개원천쏘프트웨어 즉 GNU GPL하에서 허가되거나 류사한 공개원천허가하에 있는 쏘프트웨어를 창조할수 있다. Qt공개원천판으로 Trolltech공개원천지원은 KDE와 같은 큰 규모의 성공적인 쏘프트웨어프로젝트를 창조하고 세계의 수천명의 개발자들이 무상으로 Qt공개원천판을 사용할수 있게 한다. Qt공개원천판은 Unix/X11, 마킨토쉬와 매몰형리눅스에서 사용할수 있다.

Qt를 사용하여 상업쏘프트웨어개발도 가능하며 그러자면 Professional 혹은 Enterprise판에 대한 상업허가를 구입해야 한다. Qt상업판을 산다면 자기가 좋아하는 가격과 사용허가에 맞게 자기의 쏘프트웨어를 팔수 있다. 수천개의 상업회사들은 Qt상업판을 사용하여 자기들이 팔 제품을 개발한다.

Qt공개원천판을 사용하면 GNU GPL이 부여하는 일정한 허가조건이 있어 GPL에 의해 담보되는 권한을 사용자들이 즐기도록 담보한다. 사용자들에게는 다음과 같은 권한이 부여된다.

① 자기의 쏘프트웨어를 임의의 목적으로 실행한다.

② 자기 쏘프트웨어의 원천코드를 얻어 학습하며 이것을 자기의 목적에 맞게 적응시킨다.

③ 자기 쏘프트웨어와 그 원천코드를 다른곳에 (같은 조항하에서) 재배포한다.

④ 자기 쏘프트웨어를 개선하거나 수정하며 이 갱신품을 공개출하한다.

이 권리는 누구에 의해 씌여졌는가에 관계없이 자기 쏘프트웨어가 기초하고있는 모든 모듈에 한해서 모든 원천코드에 적용된다. 또한 이 권리는 임의의 련관된 대면부정의 파일들에 적용되며 지어는 실행파일의 콤파일과 설치를 조종하는데 쓰이는 스크립트와

조종파일들도 포함한다. 그렇지 않으면 사용자들은 자기의 권리를 실현할수 없다.

이것은 자기의 쏘프트웨어가 쏘프트웨어특허, 상업적허가동의, 저작권이 있는 대면부정의 혹은 임의의 부류의 공개되지 않은 동의를 비롯하여(하지만 여기에 제한되지 않는다) GNU GPL의 조항들에 모순되는 조건들을 강요하는 모듈로 구축되여야 한다면 Qt공개원천판을 사용할수 없다는것을 의미한다. 이러한 조건에서는 Qt상업판을 리용해야 한다. (Qt공개원천판허가와 그와 관련된 질문에 대한 대답은 부록1을 참고하시오.)

제3절. Qt의 간단한 력사

Qt도구묶음은 1995년 5월에 처음으로 공개되였다. Qt는 초기에 하바드 노드(Trolltech의 CEO)와 에이리크 캄베엥(Trolltech의 사장)에 의해 개발되였다.

C++ GUI개발에 대한 하바드의 관심은 Swedish회사로부터 C++ GUI도구묶음을 설계하고 실현할데 대한 위임을 받은 1988년에 시작되였다. 2년이 지난 1990년 여름 하바드와 에이리크는 Unix와 Macintosh, Windows의 GUI에서 실행해야 하는 초음파화상용의 C++자료기지응용프로그람에 대하여 함께 연구하고있었다. 그들은 객체지향형의 체계가 필요하다고 결론하고 객체지향형 여러가동환경 GUI도구묶음을 위한 지능적인 기초를 쌓고 그것을 곧 구축하기로 하였다.

1991년에 하바드는 현재 Qt로 된 클라스들을 에이리크와 협동하여 쓰기 시작하였다. 다음 해에 에이리크는 간단하면서도 강력한 GUI프로그람작성원형인 《신호와 처리부》라는 개념을 내놓았다. 하바드는 그 개념에 동감하고 코드의 실현을 계속하였다. 1993년에 하바드와 에이리크는 Qt의 첫 그라픽스핵심을 개발하고 자체의 창문부품들을 실현하였다. 그해 말에 하바드는 《세계에서 가장 좋은 C++ GUI도구묶음》을 구축하기 위한 사업을 시작할것을 제안하였다.

그들은 하바드의 Emacs서체에서 Q가 아름답다고 보고 클라스의 앞붙이로 선택하였다. t는 X Toolkit에서 Xt에서와 같이 toolkit라는 의미에서 추가하였다. 회사는 1994년 3월 4일에 처음에 Quasar Technologies주식회사로 되였으며 그다음 Troll Tech, 오늘은 Trolltech로 되였다.

1995년 4월에 하바드종합대학의 한 교수를 통하여 접촉이 이루어져 노르웨이회사 Metis가 그들에게 Qt에 기초하여 쏘프트웨어를 개발할 계약을 하였다. 이때 Trolltech는 안트 굴브랜드센을 고용하였다. 그는 Qt의 코드에 기여하는것은 물론 훌륭한 문서체계를 실현하였다.

1995년 5월 20일에 Qt 0.90을 sunsite.unc.edu에 적재하였다. 6일후에 출하판이 comp.os.linux.announce에 발표되였다. 이것은 Qt의 처음으로 되는 공개출하판이다. Qt는 Windows와 Unix개발에 사용할수 있으며 두 가동환경에서 같은 API를 제공한다. Qt는 두가지 사용허가하에서 사용할수 있다. 상업허가는 상업적인 개발에 요구되며 무료쏘프트웨어판(free software edition)은 공개쏘프트웨어개발에 필요하다. Metis계약은 Trolltech가 크게 활약하게 하였고 10개월이상 누구도 상업용의 Qt허가를 구입하지 않았다.

1996년 3월에 유럽우주기구(European Space Agency)는 Qt의 두번째 손님으로 되였으며 10개의 상업허가권을 구입하였다. 확신을 가지고 에이리크와 하바드는 다른 개발자를 고용하였다. Qt 0.97은 5월말에 출하되였으며 1996년 9월 24일 Qt 1.0이 나왔다. 그해 말에 Qt는 1.1판에 이르고 여러 나라의 8명의 손님이 18개의 허가권을 구입하였다. 그해에 또한 마티아스 에트리치가 지도하는 KDE설계의 기초가 구축되였다.

Qt 1.2는 1997년 4월에 출하되였다. 마티아스 에트리치에 의하여 Qt를 리용하여 KDE를 구축할데 대한 론의는 Linux에서 C++ GUI개발을 위한 실제상의 표준으로 되

였다. Qt 1.3은 1997년 9월에 출하되였다.

마티아스는 1998년에 Trolltech에 합류하였으며 Qt 1의 마지막 판 1.40이 그해 9월에 개발되였다. Qt 2.0은 1999년 6월에 출하되였다. Qt 2는 주요한 구조변경을 하였으며 이전보다 훨씬 더 강력하고 성숙되였다. 또한 40개의 새 클라스와 유니코드를 추가하였다. Qt 2는 새로운 공개원천사용허가 즉 Q공개허가(Q Public License, QPL)를 가지게 되였으며 이것은 공개원천정의에 따라 콤파일된다. 1999년 8월 Qt는 Linux세계에서 가장 좋은 서고, 도구로 인정되였다. 그 무렵에 Trolltech Pty Ltd (오스트랄리아)가 수립되였다.

2000년에 Trolltech는 Qt/Embedded를 출하하였다. 이것은 매몰형Linux장치들에서 실행하도록 설계되였으며 자체의 창문체계를 제공함으로써 X11의 간단한 교체를 가능하게 한것이다. Qt/Embedded와 Qt/X11은 둘다 상업허가하에서는 물론 널리 리용된 GNU GPL(General Public License)하에서 현재 제공되였다. 2000년 말 Trolltech는 Trolltech Inc회사를 세우고 주머니형콤퓨터의 환경인 Qtopia의 초판을 출하하였다. Qt/Embedded는 2001년과 2002년에 Linux세계에서 《가장 좋은 매몰형 Linux제품》으로 알려졌다.

Qt 3.0은 2001년에 출하되였다. 현재 Qt는 Windows와 Unix, Linux, Embedded Linux, Mac OS X에서 사용할수 있게 되였다. Qt 3.0은 42개의 새 클라스들과 50만행이상의 코드를 제공하였다. Qt 3.0은 2002년에 쏘프트웨어개발상 《Jolt Productivity Award》를 받았다.

Trolltech의 판매는 회사의 출현후 해마다 배로 늘어났다. 이러한 성과는 Qt의 질과 그것을 얼마나 즐기고 있는가에 대한 반영이다. 회사의 존재, 판매, 시장의 대부분은 두 사람에 의해 조종된다. 10년이 되기전에 Qt는 《신비한》 제품으로 되였고 선발된 전문가그룹으로 알려져있으며 수천명의 손님과 수만명의 공개원천개발자들을 세계 각지에 가지고있다.

제2장. Qt의 시작

제1절. Qt학습방법

이 책에서는 독자들이 이미 C++를 알고있는것을 전제로 한다.

Qt를 배우는 가장 좋은 방법은 《Qt프로그람개발법》을 읽는것이다. 이 책은 "Hello Qt"로부터 시작하여 다중스레드프로그람작성, 2차원과 3차원 도형처리, 망프로그람작성, 그리고 XML과 같은 고급한 특성에 이르기까지 포괄적인 Qt프로그람작성방법을 제공한다.

그렇지 않으면 이 책에서 제안하는 학습프로그람을 읽는다. 설계도구없이 자기 대면부를 코드로 설계하는 순수 C++프로그람을 작성하려고 한다면 2절과 3절을 읽는다. 2절은 코드를 강조하면서 Qt프로그람작성법에 대하여 설명하였다. 3절은 차림표, 도구띠, 파일적재와 보존, 대화칸 등의 코드작성법을 보여주는 더 실천적인 실례들을 제시한다.

설계도구를 사용하여 사용자대면부를 설계하려고 한다면 《Qt프로그람개발도구》 1장의 처음 몇개 절을 읽는다. 그다음 2절과 3절을 실행하여본다.

그러면 Qt로 작은 작업프로그람을 생성하고 프로그람작성법에 대한 폭넓은 지식을 얻을수 있다. 자체로 프로젝트에 대한 작업을 직접 시작할수 있지만 Qt를 더 깊이 리해하기 위하여 3장의 1절, 3절(Qt 객체모형과 신호와 처리부)을 읽는다.

이 시점에서 4장에서 자기 프로젝트와 관련한것들을 읽을것을 권고한다. 또한 자기 프로젝트와 공통성을 가지는 실례들의 원천코드를 보는것이 좋다. 또한 Qt의 원천코드를 읽을수 있다.

$QTDIR/examples/demo에 있는 demo프로그람을 실행한다면 동작하는 Qt창문부품들을 많이 볼수 있다.

Qt는 포괄적인 문서를 제공하며 그것은 초본문 교차참고형식을 가지므로 요구하는것을 선택하여 볼수 있다. 사용자가 리용하는 문서부분은 아마 API참고일것이다. 각 련결은 API Reference를 항행하는 각이한 방법을 제공한다. 또한 Qt Assistant도구는 Qt에 제공되는데 모든 Qt API에 대한 호출을 제공하며 완전본문탐색편의를 제공한다.

제2절. 포쓰기유희의 작성

이 절에서는 Qt개발도구를 사용한 GUI프로그람작성법을 소개한다.

처음에 10행의 hello-world로 시작하여 여러가지 개념들을 소개하며 마지막에는 650행의 유희로 전환된다.

여기서 소개하는 작은 유희는 현대적인 GUI응용프로그람은 아니고 다만 몇가지 GUI수법을 리용할뿐이다. 3절에서는 좀 더 형식적이고 차림표띠, 도구띠, 적재와 보관, 대화칸와 같은 응용프로그람의 전형적인 기능들을 설명한다.

1. Hello, World!프로그람

첫 프로그람은 간단한 hello-world실레이다. 여기에는 Qt응용프로그람을 작성하고 실행하는데 필요한 최소의 골격이 들어있다. 그림 2-1은 이 프로그람의 실행화면이다.

그림 2-1. Hello, World!프로그람

```
#include <qapplication.h>
#include <qpushbutton.h>

int main( int argc, char **argv )
{
 QApplication a( argc, argv );

 QPushButton hello( "Hello world!", 0 );
 hello.resize( 100, 30 );

 a.setMainWidget( &hello );
 hello.show();
 return a.exec();
}
```

(1) 명령문설명

```
#include <qapplication.h>
```

이 행은 QApplication클라스정의를 포함한다. Qt를 사용하는 모든 응용프로그람에는 정확히 하나의 QApplication객체가 있어야 한다. QApplication은 기정서체, 유표와 같은 응용프로그람의 각종 자원을 관리한다.

```
#include <qpushbutton.h>
```

이 행은 QPushButton클라스정의를 포함한다. 각 클라스에 대한 머리부파일을 클라스를 사용하는 파일의 선두에 서술한다.

QPushButton은 사용자가 눌렀다 놓을수 있는 전형적인 GUI누름단추이다. 이것은 다른 모든 QWidget처럼 그 자체의 형식을 관리한다. 창문부품은 사용자입력을 처리하고 도형을 그릴수 있는 사용자대면부이다. 프로그람작성자는 창문부품의 내용은 물론 색과 같은 그의 총체적인 형식 그리고 많은 속성들을 변경할수 있다. QPushButton은 본문이나 QPixmap를 표시할수 있다.

```
int main( int argc, char **argv )
{
```

main()함수는 프로그람의 입구점이다. Qt를 사용할 때 항상 main()은 Qt서고에 조종을 넘기여 프로그람이 사건을 통하여 사용자의 동작에 대하여 알리기전에 몇가지 초기화를 수행해야 한다.

argc는 지령행인수의 개수이고 argv는 지령행인수의 배렬이다. 이것은 C/C++의 특성으로서 Qt에 고유하지 않지만 Qt는 이 인수들을 처리해야 한다.

```
QApplication a( argc, argv );
```

a는 이 프로그람의 QApplication객체이다. 여기서는 X Window에서 -display와 같은 몇가지 지령행인수를 생성하고 처리한다. Qt가 인식한 모든 지령행인수는 argv에서 삭제되므로 argc는 감소된다(QApplication::argv ()참고).

알아두기: QApplication객체는 Qt의 창문부분품을 사용하기전에 생성되여야 한다.

```
QPushButton hello( "Hello world!", 0 );
```

여기서 QApplication뒤에 처음의 창문체계코드가 오며 하나의 누름단추가 생성된다.

본문 "Hello world!"를 표시하기 위하여 단추를 설치하며 그 자체는 창문이다. 구성자가 부모창문으로서 0을 지정하므로 단추는 그 안에 배치되여야 한다.

```
hello.resize( 100, 30 );
```

단추는 100화소폭과 30화소높이로 설정된다. 여기에 창문의 틀이 첨부된다. 이 경우에 단추의 위치를 설정하지 않으면 기정값을 받아들인다.

```
a.setMainWidget( &hello );
```

누름단추는 응용프로그람의 기본창문부품으로서 선택된다. 사용자가 기본창문부품을 닫으면 응용프로그람은 완료한다.

기본창문부품은 없지만 대부분의 프로그람은 그것을 가진다.

```
hello.show();
```

창문부품은 그것을 창조한 다음 show()를 호출하여 표시한다.

```
return a.exec();
}
```

여기서 main()은 Qt에로 조종을 넘기며 exec()는 응용프로그람이 완료할 때 되돌아간다.

exec()에서 Qt는 사용자와 체계의 사건들을 받아들이고 처리하며 이것들을 적당한 창문부품에 넘긴다.

이제는 프로그람을 콤파일하고 실행할수 있다.

(2) 콤파일

C++응용프로그람을 콤파일하려면 makefile을 생성하여야 한다. Qt용의 makefile을 생성하는 가장 간단한 방법은 Qt에 제공되는 qmake구축도구를 사용하는것이다. 자기 등록부에 main.cpp를 보관하였다면 다음과 같이 실행한다.

```
qmake -project
qmake
```

첫째 지령은 qmake가 .pro(프로젝트)파일을 만들게 한다. 둘째 지령은 프로젝트파일에 기초하여 가동환경에 고유한 makefile을 생성한다. 이제는 make(Visual Studio를 사용한다면 nmake)라고 입력하고 처음의 Qt응용프로그람을 실행한다.

(3) 동작

프로그람을 실행하면 하나의 단추로 꽉찬 하나의 자그마한 창문을 보게 된다. 창문에서 단어 Hello World!를 읽을수 있다.

(4) 련습

창문의 크기를 조절해보시오. 단추를 눌러보시오. X Window에서 실행한다면 -geometry선택(례를 들면 -geometry 100×200+10+20)으로 프로그람을 실행하시오.

2. Quit호출

소절 1에서 창문을 창조하였으므로 사용자에 의해 응용프로그람을 적당히 완료할수 있다.

또한 기정보다 더 좋은 서체를 사용할수 있다.

그림 2-2. Quit호출

```
#include <qapplication.h>
#include <qpushbutton.h>
#include <qfont.h>

int main( int argc, char **argv )
{
 QApplication a( argc, argv );
```

```
 QPushButton quit( "Quit", 0 );
 quit.resize( 75, 30 );
 quit.setFont( QFont( "Times", 18, QFont::Bold ) );

 QObject::connect( &quit, SIGNAL(clicked()), &a, SLOT(quit()) );

 a.setMainWidget( &quit );
 quit.show();
 return a.exec();
}
```

(1) 명령문설명

```
#include <qfont.h>
```

프로그람이 QFont를 사용하므로 qfont.h를 포함하여야 한다. Qt의 서체추상화는 X에서 제공하는것과 다르고 서체의 적재와 사용은 고도로 최적화된다.

```
QPushButton quit( "Quit", 0 );
```

단추는 Quit를 통보하는데 그것은 사용자가 단추를 눌렀다 놓을 때 프로그람이 수행하는 동작이다. 여전히 부모창문으로서 0을 넘기므로 단추는 제일웃준위창문으로 된다.

```
quit.resize( 75, 30 );
```

본문이 "Hello world!"보다 짧으므로 단추의 크기를 다르게 선택한다. 또한 QFontMetrics를 사용하여 정확한 크기로 설정한다.

```
quit.setFont( QFont( "Times", 18, QFont::Bold ) );
```

여기서는 Times계렬로부터 18point강조서체를 단추의 새로운 서체로 선택한다. 즉시 서체를 생성하는것은 아니다.

또한 QApplication::setFont ()를 사용하여 전체 응용프로그람에 대하여 기정서체를 변경할수 있다.

```
QObject::connect( &quit, SIGNAL(clicked()), &a, SLOT(quit()) );
```

connect()는 Qt의 가장 주요한 특성이다. connect()는 QObject에서 정적함수이다. 이 함수를 소케트서고의 connect()함수와 삭갈리지 말아야 한다.

이 행은 2개의 Qt객체(직접적으로나 간접적으로 QObject를 계승하는 객체)들사이에 한통로련결을 확립한다. 매개 Qt객체는 (통보를 보내기 위한) 2개의 신호와 (통보를 받기 위한) 처리부를 가진다. 모든 창문부품은 Qt객체이다. 창문부품들은 QObject를 계승하는 QWidget로부터 파생된다.

여기서 quit의 clicked*()*신호는 a의 quit*()*처리부에 련결되므로 단추를 눌렀다놓으면 응용프로그람은 완료한다.(3장3절에서 자세히 설명한다.)

(2) 동작

이 프로그람을 실행하면 소절 1에서보다 아주 작은 단추가 들어있는 창문을 볼수 있다.

(3) 련습

창문의 크기를 조절해보시오. 단추를 눌러보시오. connect()는 좀 차이난다.

Quit(중지)와 련결할수 있는 다른 신호가 QPushButton에 있는가?

(암시: QPushButton은 QButton으로부터 그의 대부분의 동작을 계승한다.)

3. 가족값

이 실례는 부모와 자식창문부품의 생성방법을 보여준다.

간단히 하나의 부모와 하나의 자식을 사용한다.

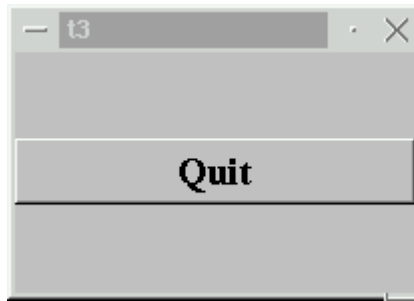



그림 2-3. 가족값

```
#include <qapplication.h>
#include <qpushbutton.h>
#include <qfont.h>
#include <qvbox.h>

int main( int argc, char **argv )
{
 QApplication a( argc, argv );

 QVBox box;
 box.resize( 200, 120 );

 QPushButton quit( "Quit", &box );
 quit.setFont( QFont( "Times", 18, QFont::Bold ) );

 QObject::connect( &quit, SIGNAL(clicked()), &a, SLOT(quit()) );

 a.setMainWidget( &box );
 box.show();

 return a.exec();
}
```

(1) 명령문설명

```
#include <qvbox.h>
```

머리부파일 qvbox.h를 추가하여 사용하려는 배치관리클라스를 얻는다.

```
QVBox box;
```

여기서는 단순히 수직칸용기를 창조한다. QVBox는 수직행안에 그 자식창문부품들을 배렬한다. 이때 매개 자식의 QWidget::sizePolicy ()에 따라 공간을 분배한다.

```
box.resize( 200, 120 );
```

그 폭은 200화소, 높이는 120화소로 설정한다.

```
QPushButton quit( "Quit", &box );
```

하나의 자식을 만든다.

본문("Quit")과 부모(box)를 가지는 QPushButton이 생성된다. 자식창문부품은 늘 부모우에 놓이며 표시될 때 부모의 경계에 의하여 잘리운다.

부모창문부품 QVBox는 자동적으로 칸의 중심에 있는 자식에 추가된다. 아무것도 추가되지 않으므로 단추는 부모가 가지고있는 모든 공간을 차지한다.

```
box.show();
```

부모창문부품이 표시될 때 그의 모든 자식들의 표시를 호출한다. (그러나 명시적인 QWidget::hide()를 수행한것들은 제외된다.)

(2) 동작

단추는 더는 전체 창문부품을 차지하지 않는다. 실제로 자연적인 크기를 얻는다. 이것은 단추크기의 암시와 크기변경방략을 사용하여 단추의 크기와 위치를 설정하는 새로운 웃준위창문부품이 있기때문이다. (QWidget::sizeHint ()와 QWidget::setSizePolicy ()참고.)

(3) 련습

창문의 크기를 변경해보시오. 단추가 어떻게 변하는가? 단추크기의 변경방략은 무엇인가? 더 큰 서체로 프로그람을 실행하면 단추의 높이에서 어떤 변화가 생기는가? 실제로 작은 창문을 만들려고 하면 어떤 현상이 생기는가?

4. 창문부품

이 실례에서는 자체의 창문부품을 만드는 방법과 창문부품의 최소 및 최대크기를 조종하는 방법을 서술하고 창문부품이름을 소개한다.

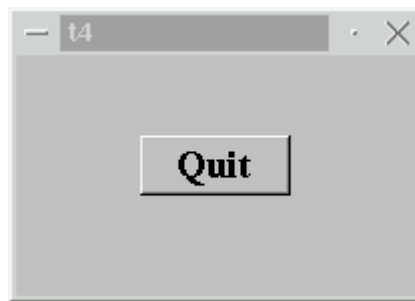



그림 2-4. 창문부품

```
#include <qapplication.h>
#include <qpushbutton.h>
#include <qfont.h>

class MyWidget : public QWidget
{
 public:
   MyWidget( QWidget *parent=0, const char *name=0 );
};

MyWidget::MyWidget( QWidget *parent, const char *name )
  : QWidget( parent, name )
{
 setMinimumSize( 200, 120 );
 setMaximumSize( 200, 120 );

 QPushButton *quit = new QPushButton( "Quit", this, "quit" );
 quit->setGeometry( 62, 40, 75, 30 );
 quit->setFont( QFont( "Times", 18, QFont::Bold ) );
```

```
 connect( quit, SIGNAL(clicked()), qApp, SLOT(quit()) );
}

int main( int argc, char **argv )
{
 QApplication a( argc, argv );

 MyWidget w;
 w.setGeometry( 100, 100, 200, 120 );
 a.setMainWidget( &w );
 w.show();
 return a.exec();
}
```

(1) 명령문설명

```
class MyWidget : public QWidget
{
  public:
   MyWidget( QWidget *parent=0, const char *name=0 );
};
```

여기서는 새로운 클라스를 만든다. 이 클라스는 QWidget를 계승하는 창문부품이고 제일 웃준위의 창문 또는 자식창문부품(제3절의 누름단추 등)일수도 있다.

이 클라스에는 오직 하나의 성원 즉 구성자(그밖에 QWidget로부터 계승되는 성원들)가 있다. 구성자는 표준 Qt창문부품구성자이고 창문부품들을 만들 때 항상 류사한 구성자를 포함해야 한다.

첫째 인수는 그 부모창문부품이다. 제일 웃준위의 창문을 만들려면 null지적자를 부모로서 지정한다. 알고있겠지만 이 창문부품은 기정으로 제일 웃준위의 창문으로 된다.

둘째 인수는 창문부품이름이다. 이것은 창문의 제목띠나 단추에 표시되는 제목이 아니며 이 창문부품을 식별하기 위한 이름이다.

```
MyWidget::MyWidget( QWidget *parent, const char *name ) :
QWidget( parent, name )
```

구성자의 실현은 여기서 시작된다. 대부분의 창문부품들처럼 구성자는 단지 부모와 이름을 QWidget구성자에 넘긴다.

```
{
 setMinimumSize( 200, 120 );
 setMaximumSize( 200, 120 );
```

이 창문부품이 크기변경방략을 모르므로 최소 및 최대크기를 꼭같이 설정한다. 다음 소절에서는 창문부품이 사용자로부터의 크기변경사건에 대응하는 방법을 보여준다.

```
 QPushButton *quit = new QPushButton( "Quit", this, "quit" );
 quit->setGeometry( 62, 40, 75, 30 );
 quit->setFont( QFont( "Times", 18, QFont::Bold ) );
```

여기에는 이름 quit를 가지는 자식창문부품(새 창문부품의 부모는 this이다.)을 만들고 설정한다. 창문부품이름은 단추본문과 관계없고 이 경우에는 우연히 비슷해졌다.

quit는 구성자의 국부변수이다. MyWidget는 그의 정보를 계속 얻지 못하지만 Qt에 의하여 MyWidget가 삭제될 때 기정으로 quit를 삭제한다. 이것은 MyWidget가 해체자를 요구하지 않기때문이다.

setGeometry()호출은 앞 절들에서 move()와 resize()가 수행하는것과 같은 일을 한다.

```
  connect( quit, SIGNAL(clicked()), qApp, SLOT(quit()) );
}
```

MyWidget클라스는 응용프로그람객체를 잘 모르므로 그것에로의 Qt지적자 qApp와 련결되여야 한다.

창문부품은 쏘프트웨어구성요소이고 될수록 일반적이고 재사용할수 있도록 하기 위하여 그 환경을 알아야 한다.

응용프로그람객체의 이름을 알고 있으면 이 원칙을 어기게 되므로 Qt는 MyWidget와 같은 구성요소가 응용프로그람객체와 대화하는 경우를 위하여 가명 qApp를 제공한다.

```
int main( int argc, char **argv )
{
   QApplication a( argc, argv );

   MyWidget w;
   w.setGeometry( 100, 100, 200, 120 );
   a.setMainWidget( &w );
   w.show();
   return a.exec();
}
```

여기서는 새로운 자식실례를 만들어 기본창문부품으로 설정하고 응용프로그람을 실행한다.

(2) 동작

이 프로그람은 이전 프로그람의 동작과 아주 류사하지만 실현방법과 동작이 약간 다르다. 그것을 보려면 크기를 변경하여야 한다.

(3) 련습

main()에 또 하나의 MyWidget객체를 만들어보시오. 어떤 일이 일어나는가?

단추를 더 많이 추가하거나 QPushButton이외의 창문부품들을 넣으시오.

5. 블로크구축

이 실례는 신호와 처리부를 사용하여 여러개의 창문부품을 만들고 모두 련결하는 방법과 크기변경사건을 조종하는 방법을 보여준다.

그림 2-5. 블로크구축

```
#include <qapplication.h>
#include <qpushbutton.h>
#include <qslider.h>
#include <qlcdnumber.h>
#include <qfont.h>
#include <qvbox.h>

class MyWidget : public QVBox
```

```
{
public:
 MyWidget( QWidget *parent=0, const char *name=0 );
};

MyWidget::MyWidget( QWidget *parent, const char *name )
  : QVBox( parent, name )
{
 QPushButton *quit = new QPushButton( "Quit", this, "quit" );
 quit->setFont( QFont( "Times", 18, QFont::Bold ) );

 connect( quit, SIGNAL(clicked()), qApp, SLOT(quit()) );

 QLCDNumber *lcd  = new QLCDNumber( 2, this, "lcd" );

 QSlider * slider = new QSlider( Horizontal, this, "slider" );
 slider->setRange( 0, 99 );
 slider->setValue( 0 );

 connect( slider, SIGNAL(valueChanged(int)), lcd,
             SLOT(display(int)) );
}

int main( int argc, char **argv )
{
 QApplication a( argc, argv );

 MyWidget w;
 a.setMainWidget( &w );
 w.show();
 return a.exec();
}
```

(1) 명령문설명

```
#include <qapplication.h>
#include <qpushbutton.h>
#include <qslider.h>
#include <qlcdnumber.h>
#include <qfont.h>
#include <qvbox.h>
```

여기서 3개의 새 머리부파일을 포함한다. qslider.h와 qlcdnumber.h는 2개의 새로운 창문부품 QSlider와 QLCDNumber를 사용하기 위하여 포함하며 qvbox.h는 Qt의 자동배치기능을 사용하기 위하여 포함한다.

```
class MyWidget : public QVBox
{
 public:
  MyWidget( QWidget *parent=0, const char *name=0 );
```

};
MyWidget::MyWidget(QWidget *parent, const char *name) : QVBox(parent, name)
{
현재 MyWidget는 QWidget대신에 QVBox에서 파생되며 QVBox의 배치관리자를 사용한다. (이것은 자식들을 수직으로 배치한다.) 또한 크기변경은 QVBox에 의해 즉 MyWidget에 의해 자동적으로 조종된다.

QLCDNumber *lcd = new QLCDNumber(2, this, "lcd");

lcd는 QLCDNumber로서 LCD방식으로 수를 표시하는 창문부품이다. 이 실례는 2자리를 표시하도록 this의 자식으로 설정된다. 그것을 "lcd"라고 이름짓는다.

QSlider * slider = new QSlider(Horizontal, this, "slider");
slider->setRange(0, 99);
slider->setValue(0);

QSlider는 전형적인 미끄럼띠로서 사용자가 창문부품을 끌기하여 일정한 범위에서 옹근수값을 조절할수 있다. 여기서는 수평미끄럼띠를 만들고 범위를 0-99로 설정하며 (QSlider::setRange ()문서 참고) 초기값을 0으로 설정한다.

connect(slider, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)));

여기서는 신호-처리부기구를 사용하여 미끄럼띠의 valueChanged()신호를 LCD수의 display()처리부에 련결한다.

미끄럼띠의 값이 바뀔 때마다 valueChanged()신호를 내보내여 새로운 값을 전송한다. 그 신호가 LCD수의 display()처리부에 련결되여있으므로 처리부는 신호가 전송될 때 호출된다. 어느 객체도 다른 객체에 대하여 모른다. 이것은 콤포넨트프로그람작성의 본질이다.

한편 처리부는 보통의 C++성원함수로서 표준 C++호출규칙을 따른다.

(2) 동작

LCD수는 미끄럼띠에 대한 사용자의 모든 조작을 반영하며 창문부품은 크기변경을 잘 조종한다. 창문크기가 변경될 때(그것이 가능하기때문에) LCD수 창문부품의 크기는 달라지지만 다른것은 거의 비슷하다.

(3) 련습

자리수를 더 많이 추가하거나 방식을 바꾸어 LCD수를 변경해보시오. 기수를 설정하기 위하여 4개의 누름단추를 추가할수 있다.

또한 미끄럼띠의 범위를 변경할수 있다. 미끄럼띠보다 QSpinBox를 사용하는것이 더 좋다. LCD수가 넘어날 때 응용프로그람을 중지한다.

6. 복잡한 블로크의 구축

이 실례는 새로운 구성요소로서 2개의 창문부품을 밀봉하는 방법과 많은 창문부품을 간단히 사용하는 방법을 보여준다. 처음으로 사용자정의창문부품을 자식창문부품으로 사용한다.

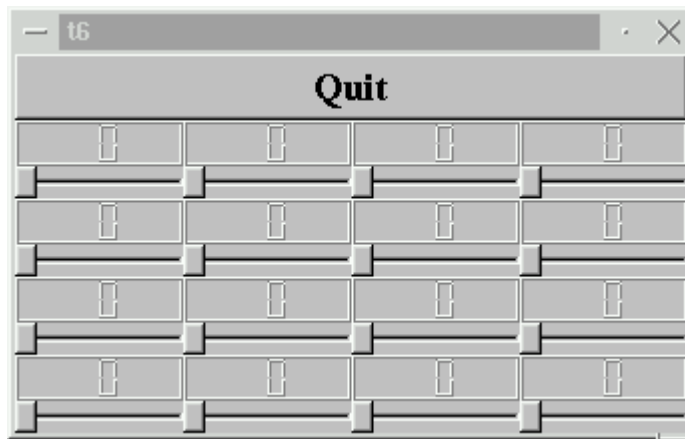



그림 2-6. 복잡한 블로크의 구축

```
#include <qapplication.h>
#include <qpushbutton.h>
#include <qslider.h>
#include <qlcdnumber.h>
#include <qfont.h>
#include <qvbox.h>
#include <qgrid.h>

class LCDRange : public QVBox
{
public:
 LCDRange( QWidget *parent=0, const char *name=0 );
};

LCDRange::LCDRange( QWidget *parent, const char *name )
  : QVBox( parent, name )
{
 QLCDNumber *lcd  = new QLCDNumber( 2, this, "lcd"  );
 QSlider * slider = new QSlider( Horizontal, this, "slider" );
 slider->setRange( 0, 99 );
 slider->setValue( 0 );
 connect( slider, SIGNAL(valueChanged(int)), lcd,
            SLOT(display(int)) );
}

class MyWidget : public QVBox
{
public:
 MyWidget( QWidget *parent=0, const char *name=0 );
};

MyWidget::MyWidget( QWidget *parent, const char *name ) :
```

```
QVBox( parent, name )
    {
     QPushButton *quit = new QPushButton( "Quit", this, "quit" );
     quit->setFont( QFont( "Times", 18, QFont::Bold ) );

     connect( quit, SIGNAL(clicked()), qApp, SLOT(quit()) );

     QGrid *grid = new QGrid( 4, this );

     for( int r = 0 ; r < 4 ; r++ )
      for( int c = 0 ; c < 4 ; c++ )
       (void)new LCDRange( grid );
    }

    int main( int argc, char **argv )
    {
     QApplication a( argc, argv );

     MyWidget w;
     a.setMainWidget( &w );
     w.show();
     return a.exec();
    }
```

(1) 명령문설명

```
class LCDRange : public QVBox
{
public:
 LCDRange( QWidget *parent=0, const char *name=0 );
};
```

LCDRange창문부품은 구성자만 가지는 창문부품이다. 창문부품의 이러한 분류는 비효과적이므로 몇가지 API를 후에 추가한다.

```
LCDRange::LCDRange( QWidget *parent, const char *name )
  : QVBox( parent, name )
{
 QLCDNumber *lcd  = new QLCDNumber( 2, this, "lcd"  );
 QSlider * slider = new QSlider( Horizontal, this, "slider" );
 slider->setRange( 0, 99 );
 slider->setValue( 0 );
 connect( slider, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)) );
}
```

이것은 소절 5에서 MyWidget구성자로부터 직접 복사한다. 유일한 차이는 단추가 없고 클라스이름이 달라진것이다.

```
class MyWidget : public QVBox
{
public:
 MyWidget( QWidget *parent=0, const char *name=0 );
```

```
};
```

MyWidget도 구성자를 제외한 어떤 API도 포함하지 않는다.

```
MyWidget::MyWidget( QWidget *parent, const char *name ) :
QVBox( parent, name )
{
  QPushButton *quit = new QPushButton( "Quit", this, "quit" );
  quit->setFont( QFont( "Times", 18, QFont::Bold ) );

  connect( quit, SIGNAL(clicked()), qApp, SLOT(quit()) );
```

하나의 Quit단추와 여러개 LCDRange객체들을 가질수 있게 지금 LCDRange에 있는 누름단추를 분리한다.

```
  QGrid *grid = new QGrid( 4, this );
```

4개 렬을 가지는 QGrid객체를 만든다. QGrid창문부품은 행과 렬에 자식들을 자동적으로 배치하고 행수 또는 렬수를 지정할수 있으며 QGrid는 새로운 자식들을 발견하고 그것들을 살창에 삽입한다.

```
  for( int r = 0 ; r < 4 ; r++ )
    for( int c = 0 ; c < 4 ; c++ )
      (void)new LCDRange( grid );
}
```

살창객체의 자식들인 4×4 LCDRange를 생성한다. QGrid창문부품은 그것들을 배치한다.

(2) 동작

이 프로그람은 많은 창문부품을 한번에 사용하는 간단한 방법을 보여준다. 매개는 앞의 소절에서 미끄럼띠와 LCD수처럼 동작한다. 차이는 실현에 있다.

(3) 련습

기동할 때 매개 미끄럼띠를 각이한 우연수로 초기화하시오.

원천은 "4"의 3개 실현값을 포함한다. QGrid구성자호출에서 그것을 변경하면 어떤 일이 생기는가? 왜 그런가?

7. 한 객체가 다른 객체를 유도하기

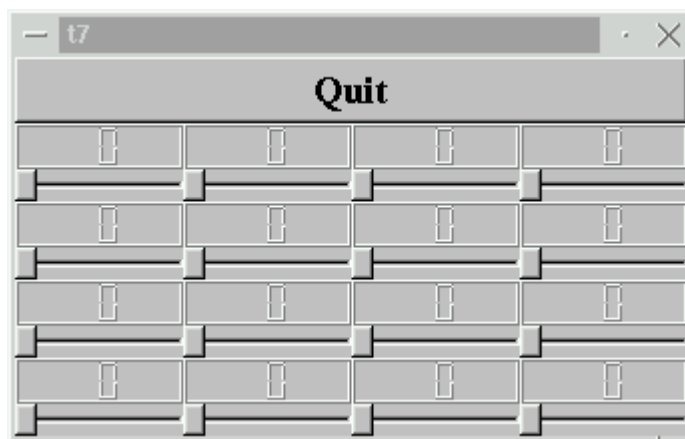



그림 2-7. 한 객체가 다른 객체를 유도하기

이 실례는 신호와 처리부를 가지는 사용자정의창문부품들을 생성하여 모두 련결하는 방법을 보여준다. 우선 t7보조등록부에 있는 여러개의 파일들에 원천을 나누어 보관한다.

- t7/lcdrange.h는 LCDRange클라스를 정의한다.
- t7/lcdrange.cpp는 LCDRange를 실현한다.
- t7/main.cpp는 MyWidget와 main을 포함한다.

(1) 명령문설명

① t7/lcdrange.h

이 파일은 주로 소절 6의 main.cpp로부터 옮겨온것으로서 달라진것만 설명한다.

```
#ifndef LCDRANGE_H
#define LCDRANGE_H
```

이것은 전형적인 C구성으로서 머리부파일을 한번이상 포함하여야 하는 경우 오유를 피하게 한다. #ifndef는 모든 머리부파일을 둘러막아야 한다.

```
#include <qvbox.h>
```

qvbox.h를 포함한다. LCDRange는 QVBox를 계승하므로 부모클라스의 머리부파일을늘 포함하여야 한다. 앞의 소절에서 qwidget.h는 qpushbutton.h와 같은 다른 머리부파일을 통하여 간접적으로 포함된다.

```
class QSlider;
```

이것은 다른 중요한 수법으로서 자주 사용된다. 클라스의 대면부에서 QSlider를 요구하지 않으므로 머리부파일에서 클라스의 앞방향선언만 사용하고 실현에서만.cpp파일에서 QSlider의 머리부파일을 포함한다.

이것은 머리부파일이 변경되였을 때 적은 파일들을 다시 콤파일하게 하므로 큰 프로젝트의 콤파일을 훨씬 빠르게 한다. 즉 큰 프로젝트의 콤파일을 2배이상 고속화할수 있다.

```
class LCDRange : public QVBox
{
 Q_OBJECT
public:
 LCDRange( QWidget *parent=0, const char *name=0 );
```

Q_OBJECT마크로는 신호나 처리부를 포함하는 모든 클라스들에 포함되여야 한다. 필요하다면 메타객체파일에서 실현되는 함수들을 정의한다.

```
 int value() const;
 public slots:
  void setValue( int );


 signals:
 void valueChanged( int );
```

이 세개 성원은 프로그람에서 창문부품들과 다른 부분품들사이의 대면부를 만든다. 지금까지 LCDRange는 실제로 대면부를 전혀 가지지 않았다.

value()는 LCDRange의 값을 호출하기 위한 공개함수이다. setValue()는 첫 사용자정의처리부이고 valueChanged()는 첫 사용자정의신호이다.

처리부들은 보통의 방법으로 실현되여야 한다.(처리부도 C++성원함수이다.) 신호는 자동적으로 메타객체파일에 실현되고 보호 C++함수들의 호출규칙을 따른다. (즉 신호는 자기가 정의되는 클라스 혹은 그 기초클라스에 의해서만 발생될수 있다.)

신호 valueChanged()는 LCDRange의 값이 변할 때 사용된다.

② t7/lcdrange.cpp

이 파일은 주로 t6/main.cpp로부터 옮기였으며 여기서는 달라진것만 설명한다.

```
connect( slider, SIGNAL(valueChanged(int)), lcd,
            SLOT(display(int)) );
connect( slider, SIGNAL(valueChanged(int)),
            SIGNAL(valueChanged(int)) );
```

이 코드는 LCDRange구성자로부터 발취한것이다.

첫째 련결은 앞의 소절에서 본것과 같다. 둘째는 새로운것이다. 이것은 미끄럼띠의 valueChanged()신호를 이 객체의 valueChanged신호에 련결한다. 3인수를 가진 connect()는 늘 이 객체에서 신호나 처리부들을 련결한다.

신호는 다른 신호들에 련결될수 있다. 첫 신호가 발생될 때 둘째 신호도 발생된다.

사용자가 미끄럼띠에 조작할 때 미끄럼띠는 그 값이 변경되면 valueChanged()신호를 발생한다. 이 신호는 QLCDNumber의 display()처리부와 LCDRange의 valueChanged()신호에 모두 련결된다.

이리히여 신호가 발생될 때 LCDRange는 그 자체의 valueChanged()신호를 발생한다. 또한 QLCDNumber::display()가 호출되여 새로운 수가 표시된다.

LCDRange::valueChanged()는 QLCDNumber::display()전후에 발생될수 있고 그 실행순서는 따로 지정된것이 없다.

```
int LCDRange::value() const
{
 return slider->value();
}
```

value()의 실현은 간단하며 단순히 미끄럼띠의 값을 돌려준다.

```
void LCDRange::setValue( int value )
{
 slider->setValue( value );
}
```

setValue()의 실현도 간단하다. 미끄럼띠와 LCD수가 련결되므로 미끄럼띠값의 설정은 자동적으로 LCD수를 갱신한다. 또한 미끄럼띠는 값이 범위를 벗어나면 자동적으로 그것을 조절한다.

③ t7/main.cpp

```
LCDRange *previous = 0;
for( int r = 0 ; r < 4 ; r++ ) {
 for( int c = 0 ; c < 4 ; c++ ) {
  LCDRange* lr = new LCDRange( grid );
  if ( previous )
   connect( lr, SIGNAL(valueChanged(int)),
      previous, SLOT(setValue(int)) );
  previous = lr;
 }
}
```

MyWidget용구성자를 제외한 main.cpp의 코드는 모두앞의 소절에서 복사하였다. 16개의 LCDRange객체를 창조할 때 현재는 신호-처리부기구를 사용하여 그것들을 련결한다. 매개 객체는 앞 객체의 setValue()처리부와 련결된 valueChanged()신호를 가진다. LCDRange는 값이 변화할 때 신호 valueChanged()를 발생하므로 여기서는

신호와 처리부들의 《사슬》을 창조한다.

(2) 콤파일

여러파일 응용프로그람용 makefile의 창조는 단일파일응용프로그람용 makefile의 창조와 다르다. 자체의 등록부에 이 실례의 모든 파일들을 보관하였다면 다음과 같이 해야 한다.

qmake -project
qmake

첫 지령은 qmake가 .pro(프로젝트)파일을 창조한다. 둘째 지령은 프로젝트파일에 기초하여 가동환경에 고유한 makefile을 창조한다. 이제는 make (혹은 Visual Studio를 사용하는 경우 nmake)를 입력하여 응용프로그람을 구축할수 있다.

(3) 동작

기동할 때 프로그람의 창문은 이전과 같다.

(4) 련습

오른쪽 아래의 미끄럼띠를 사용하여 모든 LCD를 50으로 설정한다. 그다음 미끄럼띠손잡이의 왼쪽을 한번 찰칵하여 우의 절반을 40으로 설정한다. 그러면 마지막으로 조작한것의 왼쪽에 있는 미끄럼띠를 사용하여 처음의 7개 LCD들을 50으로 설정한다.

오른쪽 아래에 있는 미끄럼띠 손잡이의 오른쪽을 찰칵한다. 어떤 일이 생기는가? 왜 이것이 정확한 동작으로 되는가?

8. 전투준비

이 실례에서는 자체로 그릴수 있는 첫 사용자정의창문부품을 소개한다. 또한 두행의 코드로 쓸모있는 건반대면부를 추가한다.

- t8/lcdrange.h는 LCDRange클라스를 정의한다.
- t8/lcdrange.cpp는 LCDRange를 실현한다.
- t8/cannon.h는 CannonField클라스를 정의한다.
- t8/cannon.cpp는 CannonField를 실현한다.
- t8/main.cpp는 MyWidget와 main을 포함한다.

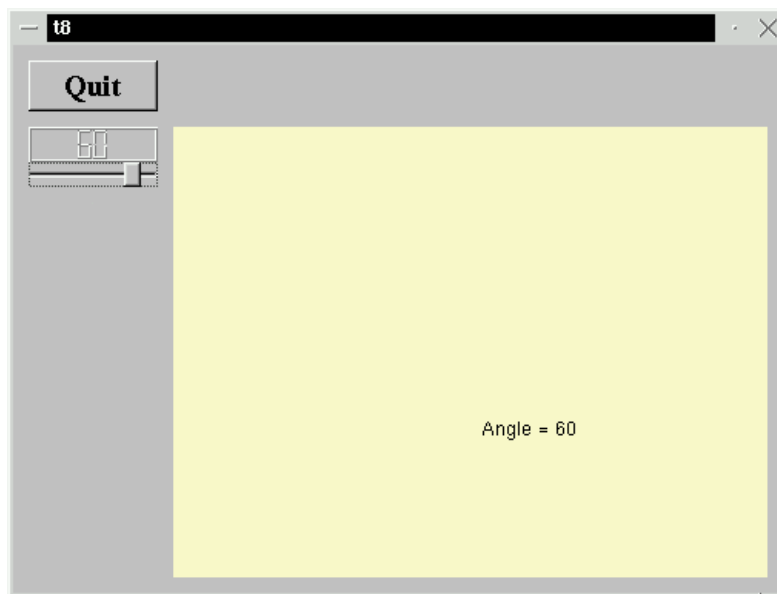



그림 2-8. 전투준비

(1) 명령문설명

① t8/lcdrange.h

이 파일은 소절 7의 lcdrange.h와 아주 비슷하다. 하나의 처리부 setRange()를 추가한다.

```
void setRange( int minVal, int maxVal );
```

이제 LCDRange의 범위설정기능을 추가한다. 현재 0~99로 고정한다.

② t8/lcdrange.cpp

구성자를 변경한다.

```
void LCDRange::setRange( int minVal, int maxVal )
{
 if ( minVal < 0 || maxVal > 99 || minVal > maxVal ) {
   qWarning( "LCDRange::setRange(%d,%d)\n"
      "\tRange must be 0..99\n"
      "\tand minVal must not be greater than maxVal", minVal,
       maxVal );
   return;
 }
 slider->setRange( minVal, maxVal );
}
```

setRange()는 LCDRange에서 미끄럼띠의 범위를 설정한다. 항상 두자리를 현시하도록 QLCDNumber를 설정하였으므로 minVal과 maxVal의 가능한 범위를 0~99로 제한하여 QLCDNumber의 자리넘침을 피하려고 한다. 인수가 옳지 않으면 Qt의 qWarning()함수에 의하여 사용자에게 경고를 내고 즉시 되돌아간다. qWarning()은 기정으로 출력을 stderr에 보내는 printf형식의 함수이다. 필요하다면 ::qInstallMsgHandler()에 의해 자체의 처리함수함수를 설치할수 있다.

③ t8/cannon.h

CannonField는 자기를 현시하는 방법을 알고있는 새로운 사용자정의창문부품이다.

```
class CannonField : public QWidget
{
  Q_OBJECT
  public:
    CannonField( QWidget *parent=0, const char *name=0 );
```

CannonField는 QWidget를 계승하며 LCDRange에서와 류사한 성원들을 사용한다.

```
 int angle() const { return ang; }
 QSizePolicy sizePolicy() const;
 public slots:
 void setAngle( int degrees );
 signals:
 void angleChanged( int );
```

CannonField는 오직 LCDRange의 값대신에 각도를 리용하는 대면부를 제공한다.

```
protected:
 void paintEvent( QPaintEvent * );
```

이것은 QWidget의 수많은 사건처리함수중에서 두번째로 만나는것이다. 이 가상함수는 창문부품이 자체를 갱신해야 할 때마다 Qt에 의해 호출된다. (즉 창문부품의 표면을 그린다.)

④ t8/cannon.cpp

```
CannonField::CannonField( QWidget *parent, const char *name )
```

```
: QWidget( parent, name )
{
```
앞의 소절에서 LCDRange와 비슷한 성원함수들을 사용한다.
```
ang = 45;
setPalette( QPalette( QColor( 250, 250, 200) ) );
}
```
구성자는 각도값을 45도로 초기화하고 이 창문부품용의 사용자정의조색판을 설정한다. 이 조색판은 지적된 색을 배경으로 사용하고 다른 색들을 적당히 선택한다. (이 창문부품에서는 오직 배경과 본문색만 실제로 사용한다.)
```
void CannonField::setAngle( int degrees )
{
 if ( degrees < 5 )
  degrees = 5;
 if ( degrees > 70 )
  degrees = 70;
 if ( ang == degrees )
  return;
 ang = degrees;
 repaint();
 emit angleChanged( ang );
}
```
이 함수는 각도값을 설정한다. 범위를 5~70으로 선택하고 그에 따라 도수를 조절한다. 새 각도가 범위를 벗어나면 경고를 내지 않도록 선택한다.

새로운 각도가 낡은 값과 같으면 즉시 되돌아간다. 각도가 실제로 변했을 때 신호 angleChanged()를 발생하는것만 중요하다.

그다음 새로운 각도값을 설정하고 창문부품을 다시 그린다. QWidget::repaint() 함수는 창문부품을 지우고(보통 배경으로 채운다.) 창문부품에 그리기사건을 보낸다. 이리하여 창문부품의 그리기사건함수호출이 발생한다.

끝으로 angleChanged()신호를 발생하여 외부에 각도가 변화하였다고 전한다. emit예약어는 Qt에만 있으며 표준 C++문법이 아니다. 사실상 그것은 마크로이다.
```
void CannonField::paintEvent( QPaintEvent * )
{
 QString s = "Angle = " + QString::number( ang );
 QPainter p( this );
 p.drawText( 200, 200, s );
}
```
이것은 그리기사건처리함수에 써넣기 위한 첫 시도이다. 사건인수는 그리기사건의 서술을 포함한다. QPaintEvent는 갱신해야 할 창문부품안의 령역을 포함한다. 전부 그리므로 시간적으로 뜨다.

코드에서는 창문부품안의 각도값을 고정위치에 현시한다. 우선 본문과 각도를 가지는 QString을 창조한 다음 이 창문부품에 조작하는 QPainter를 창조하고 그것을 리용하여 문자렬을 그린다. 후에 QPainter에서 많은 일을 수행한다.

⑤ t8/main.cpp
```
#include "cannon.h"
```
새로운 클라스를 포함한다.
```
class MyWidget: public QWidget
```

```
{
public:
  MyWidget( QWidget *parent=0, const char *name=0 );
};
```

이번에는 하나의 LCDRange와 하나의 CannonField를 제일웃준위창문부품에 포함한다.

```
LCDRange *angle = new LCDRange( this, "angle" );
```

구성자에서는 LCDRange를 창조하고 설정한다.

```
angle->setRange( 5, 70 );
```

LCDRange의 범위를5~70도로 설정한다.

```
CannonField *cannonField = new CannonField( this, "cannonField" );
```

CannonField를 창조한다.

```
connect( angle, SIGNAL(valueChanged(int)), cannonField, SLOT(setAngle(int)) );
connect( cannonField, SIGNAL(angleChanged(int)), angle, SLOT(setValue(int)) );
```

여기서는 CannonField의 setAngle()처리부에 LCDRange의 valueChanged()신호를 련결한다. 이것은 사용자가 LCDRange를 조작할 때마다 CannonField의 각도값을 갱신한다. 또한 반대련결을 만들어 CannonField에서의 각도변경이 LCDRange값을 갱신하게 한다. 실례에서는 CannonField의 각도를 직접 변경하지 않지만 마지막 connect()를 수행하여 앞으로의 변경이 그 두개 값들사이의 동기화를 파괴하지 않는다는것을 담보한다.

이것은 콤퓨넨트프로그람작성법의 능력과 적당한 밀봉성을 설명한다.

실제로 각도가 변경될 때만 angleChanged()신호를 발생하는것이 얼마나 중요한가를 알아야 한다. LCDRange와 CannonField가 둘다 이 검사를 생략하면 프로그람은 값들중 하나의 첫 변경에 대하여 무한순환에 들어간다.

```
QGridLayout *grid = new QGridLayout( this, 2, 2, 10 );
//2x2, 10 pixel border
```

지금까지는 조립이 필요하지 않은 QVBox와 QGrid창문부품들을 기하학적관리에 사용하였다. 그러나 이제는 배치관리자를 좀 더 조절하여 강력한 QGridLayout클라스로 절환한다. QGridLayout는 창문부품이 아니고 임의의 창문부품의 자식들을 관리할 수 있는 다른 클라스이다.

설명문에서 보여주는것처럼 10화소테두리를 가지는 2×2배렬을 창조한다. (QGridLayout용 구성자는 좀 신비스러우므로 설명문을 넣는것이 좋다.)

```
grid->addWidget( quit, 0, 0 );
```

살창의 제일왼쪽세포 0, 0에 Quit단추를 추가한다.

```
grid->addWidget( angle, 1, 0, Qt::AlignTop );
```

세포의 꼭대기와 일직선되는 왼쪽바닥세포에 각도 LCDRange를 놓는다. (이러한 배렬은 QGridLayout는 허용하지만 QGrid는 허용하지 않는것들중의 하나이다.)

```
grid->addWidget( cannonField, 1, 1 );
```

오른쪽아래세포에 CannonField객체를 놓는다. (오른쪽웃끝세포는 비여있다.)

```
grid->setColStretch( 1, 10 );
```

오른쪽란(렬1)을 신축할수 있다는것을 QGridLayout에게 알린다. 왼쪽란이 없으므로(그것은 기정신축결수 0을 가진다.) QGridLayout는 왼쪽창문부품들의 크기가 변경되지 않게 하려고 하며 MyWidget의 크기가 조절될 때 CannonField를 조절한다.

angle->setValue(60);

초기각도값을 설정한다. 이것은 련결을 LCDRange로부터 CannonField로 절환한다.

angle->setFocus();

마지막으로 할 일은 각도가 건반초점을 가지도록 설정하여 건반입력이 기정으로 LCDRange창문부품으로 가도록 하는것이다.

LCDRange는 어떤 keyPressEvent()도 포함하지 않으므로 그리 쓸모있지 않는다. 그러나 구성자는 새 행을 얻는다.

setFocusProxy(slider);

LCDRange는 미끄럼띠를 그 초점대리로 설정한다. 이것은 프로그람 혹은 사용자가 LCDRange건반초점을 주려고 할 때 미끄럼띠가 그것을 고려해야 한다는것을 의미한다. QSlider는 괜찮은 건반대면부를 가지므로 바로 한행의 코드에 의해 LCDRange의 대면부를 주었다.

(2) 동작

지금 건반은 동작하며 방향건, Home, End, PageUp와 PageDown은 모두 막연하게 느낄수 있는 일을 수행한다.

미끄럼띠가 조작되면 CannonField는 새로운 각도값을 현시한다. 크기조절할 때 CannonField는 될수록 많은 공간을 차지한다.

8bit현시기를 가지는 Windows콤퓨터들에서 새로운 배경색이 떨린다. 다음 항은 이에 대하여 작업한다.

(3) 련습

창문의 크기를 조절하시오. 창문을 줄이면 어떤 일이 생기는가?

AlignTop를 삭제하면 LCDRange의 위치와 크기가 어떻게 되는가? 무엇때문인가?

왼쪽란에 령아닌 신축결수를 준다면 창문크기를 조절할 때 어떤 현상이 생기는가?

QButton::setText()호출에서 "Quit"를 "&Quit"로 변경해보시오. 단추의 형식이 어떻게 변하는가? 프로그람을 실행할 때 Alt+Q(일부 건반에서 Meta+Q)를 누르면 어떤 일이 생기는가?

본문을 CannonField의 중심에 배치하시오.

9. 포쏘기

이 실례에서는 작고 매력있는 청색포를 그린다. cannon.cpp만 앞항과 다르다.

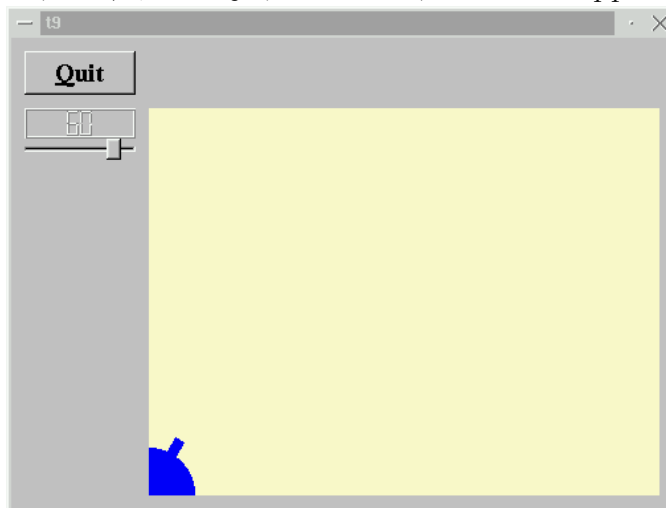



그림 2-9. 포쏘기

- t9/lcdrange.h는 LCDRange클라스를 정의한다.
- t9/lcdrange.cpp는 LCDRange실현을 포함한다.
- t9/cannon.h는 CannonField클라스를 정의한다.
- t9/cannon.cpp는 CannonField실현을 포함한다.
- t9/main.cpp는 MyWidget와 main을 포함한다.

(1) 명령문설명

① t9/cannon.cpp

```
void CannonField::paintEvent( QPaintEvent * )
{
  QPainter p( this );
```

이제는 QPainter를 진지하게 사용한다. 이 창문부품에 조작하는 그리기객체(painter)을 창조한다.

```
  p.setBrush( blue );
```

QPainter는 직4각형, 원 혹은 다른것을 색칠할 때 자기 솔을 사용한다. 여기서는 청색솔을 사용하도록 설정한다. (또한 패턴을 사용할수 있다.)

```
  p.setPen( NoPen );
```

그리고 QPainter가 테두리를 그릴 때는 펜을 사용한다. 여기서는 NoPen으로 설정하는데 이것은 그릴 때 테두리가 없으며 청색솔을 사용한다는것을 의미한다.

```
  p.translate( 0, rect().bottom() );
```

QPainter::translate()함수는 QPainter의 자리표계를 변환한다. 즉 변위만큼 그것을 이동한다. 여기서는 창문부품의 왼쪽아래구석을 점 (0, 0)으로 설정한다. x와 y방향은 변하지 않는다. 즉 창문부품안의 모든 y자리표는 현재 부수이다(Qt의 자리표계에 대한 자세한 정보는 5장 3절 참고).

```
  p.drawPie( QRect(-35, -35, 70, 70), 0, 90*16 );
```

drawPie()함수는 시작각도와 호의 길이를 리용하여 특정한 직4각형안에 부채형을 그린다. 각도는 도의 1/16로 지정된다. 령도는 3시위치, 그리기방향은 시계바늘과 반대방향이다. 여기서는 창문부품의 왼쪽아래구석의 4분의 1을 그린다. 부채형은 청색으로 도색되고 테두리선이 없다.

```
  p.rotate( -ang );
```

QPainter::rotate()함수는 원점을 중심으로 QPainter의 자리표계를 회전시킨다. 회전인수는 도수로 주어진 float이며(우와 같이 도의 1/16로 주어지지 않는다.) 시계바늘과 반대방향이다. 여기서는 자리표계를 시계바늘이 도는 방향과 반대로 ang도만큼 회전시킨다.

```
  p.drawRect( QRect(33, -4, 15, 8) );
```

QPainter::drawRect()함수는 지정된 직4각형을 그린다. 여기서는 포신을 그린다.

우와 같이 자리표계가 변하였을 때(변환, 회전, 확대축소, 혹은 잘라냈을 때) 결과를 상상하기 힘들수 있다.

이 경우에 우선 자리표계가 변환되고 그다음 회전된다. 직4각형 QRect(33, -4, 15, 8)이 변환된 자리표계에서 그려지면 다음과 같이 보일수 있다.

직4각형이 CannonField창문부품의 경계에 의해 잘리운다. 자리표계를 가령 60도 회전할 때 직4각형은 (0, 0)를 중심으로 회전하며 자리표계를 변환하였으므로 이것은 왼쪽아래구석이다. 결과는 다음과 같다.

이번에는 창문이 왜 떨리지 않았는가만 설명하지 않았다.

```
int main( int argc, char **argv )
{
  QApplication::setColorSpec( QApplication::CustomColor );
  QApplication a( argc, argv );
```

이 프로그람에 다른 색할당전략을 사용한다고 Qt에게 알린다. 단일하고 정확한 색할당전략은 없다. 이 프로그람이 수많은 색이 아니라 황색을 사용하므로 CustomColor가 제일 좋다. 몇가지 할당방식이 있으며 QApplication::setColorSpec()문서에서 그것들을 읽을수 있다.

대체로 기정이 좋으므로 이것을 무시한다. 우연적으로 비일반색을 사용하는 응용프로그람들은 나쁘게 보이며 할당전략의 변경이 흔히 그 프로그람들을 방조한다.

(2) 동작

미끄럼띠를 조작할 때 그려진 포의 각도가 그에 따라 변한다.

Quit단추의 Q에는 현재 밑줄이 놓이고 Alt+Q는 중지작용을 수행한다. 그 리유를 모르면 8.의 련습을 하지 않았기때문이다.

느린 콤퓨터에서 포가 특별히 깜빡거린다는것을 알수 있다. 다음 항에서 이것을 고착시킨다.

(3) 련습

NoPen대신에 다른 펜을 설정하시오. 패턴솔을 설정하시오.

단추본문을 "&Quit"대신에 "Q&uit" 혹은 "Qu&it"로 하시오. 어떤 일이 생기는가?

10. 명주처럼 부드럽게

이 실례에서는 깜빡거림을 없애기 위한 픽스매프그리기를 소개한다. 또한 힘조종요소를 추가한다.

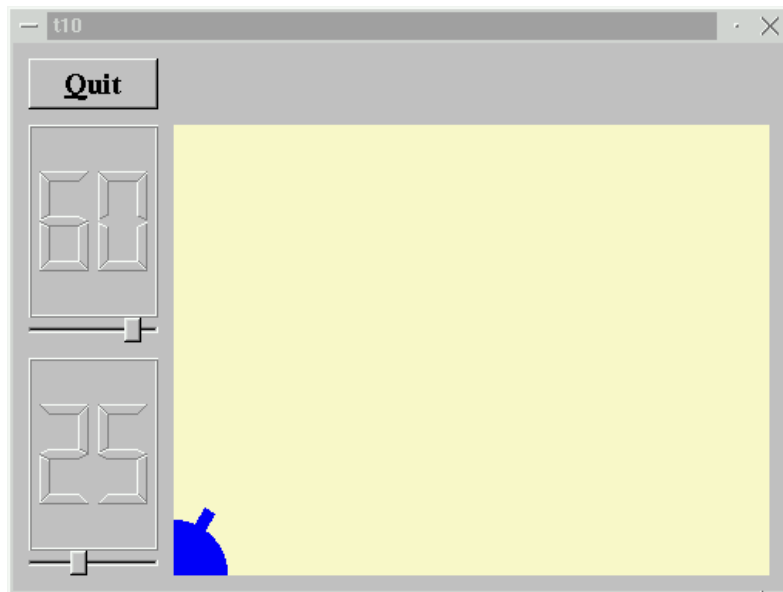



그림 2-10. 힘조종요소의 추가

• t10/lcdrange.h는 LCDRange클라스를 정의한다.
• t10/lcdrange.cpp는 LCDRange실현을 포함한다.
• t10/cannon.h는 CannonField클라스를 정의한다.
• t10/cannon.cpp는 CannonField실현을 포함한다.
• t10/main.cpp는 MyWidget와 main을 포함한다.

(1) 명령문설명

① t10/cannon.h

이제 CannonField는 각도와 함께 힘값을 가진다.

```
 int   angle() const { return ang; }
 int   force() const { return f; }

public slots:
 void  setAngle( int degrees );
 void  setForce( int newton );

signals:
 void  angleChanged( int );
 void  forceChanged( int );
```

힘에 대한 대면부는 실제상 각도에서와 같다.

```
private:
 QRect cannonRect() const;
```

개별적인 함수안에 포를 둘러싸는 직4각형의 정의를 넣는다.

```
 int ang;
 int f;
};
```

힘은 f에 보관된다.

② t10/cannon.cpp

```
#include <qpixmap.h>
```

QPixmap클라스를 정의한다.

```
CannonField::CannonField( QWidget *parent, const char *name )
  : QWidget( parent, name )
{
 ang = 45;
 f = 0;
 setPalette( QPalette( QColor( 250, 250, 200) ) );
}
```

힘(f)은 0으로 초기화된다.

```
void CannonField::setAngle( int degrees )
{
 if ( degrees < 5 )
  degrees = 5;
 if ( degrees > 70 )
  degrees = 70;
 if ( ang == degrees )
  return;
```

```
  ang = degrees;
  repaint( cannonRect(), FALSE );
  emit angleChanged( ang );
 }
```
setAngle()함수를 약간 변경하고 포를 포함하는 창문부품부분만 다시 그린다. FALSE인수는 그리기사건이 창문부품에 보내지기전에 지정된 직4각형을 지우지 말아야 한다는것을 지정한다. 이것은 그리기를 좀 더 빠르고 유연하게 한다.
```
 void CannonField::setForce( int newton )
 {
  if ( newton < 0 )
   newton = 0;
  if ( f == newton )
   return;
  f = newton;
  emit forceChanged( f );
 }
```
setForce()의 실현은 setAngle()과 비슷하다. 유일한 차이는 힘값을 표시하지 않으므로 창문부품을 다시 그릴 필요가 없는것이다.
```
 void CannonField::paintEvent( QPaintEvent *e )
 {
  if ( !e->rect().intersects( cannonRect() ) )
   return;
```
이제는 갱신이 요구되는 창문부품부분만 다시 그리도록 그리기사건을 최적화한다. 우선 무엇인가 그려야 하는가 검사하고 그렇지 않으면 되돌아간다.
```
  QRect cr = cannonRect();
  QPixmap pix( cr.size() );
```
그다음 깜빡거림이 없는 그리기에 사용할 일시픽스매프를 창조한다. 모든 그리기조작은 이 픽스매프에 수행된 다음 픽스매프는 단일조작으로 화면에 그려진다.

이것은 깜빡거림이 없는 그리기의 본질이다. 즉 매개 화소에 정확히 한번 그린다. 드문히 그리기오유가 생기고 깜빡거림이 많이 생긴다. 이 실례에서는 문제가 없다. 코드를 썼을 때 여전히 깜빡거림때문에 콤퓨터가 느리다.
```
  pix.fill( this, cr.topLeft() );
```
이 창문부품으로부터 픽스매프를 배경으로 채운다.
```
  QPainter p( &pix );
  p.setBrush( blue );
  p.setPen( NoPen );
  p.translate( 0, pix.height() - 1 );
  p.drawPie( QRect( -35,-35, 70, 70 ), 0, 90*16 );
  p.rotate( -ang );
  p.drawRect( QRect(33, -4, 15, 8) );
  p.end();
```
소절 9에서처럼 그리지만 현재 픽스매프안에 그린다.

이 시점에서는 그리기객체와 픽스매프를 가지지만 여전히 화면에 그려지지 않는다.
```
  p.begin( this );
  p.drawPixmap( cr.topLeft(), pix );
```
그러므로 CannonField자체가 그리기객체를 열고 픽스매프를 그린다.

꼭대기의 두행과 아래의 두행 그리고 깜빡거림은 전혀 없다.

```
QRect CannonField::cannonRect() const
{
 QRect r( 0, 0, 50, 50 );
 r.moveBottomLeft( rect().bottomLeft() );
 return r;
}
```

이 함수는 창문부품자리표계에서 포를 둘러싸는 직4각형을 돌려준다. 우선 50×50크기를 가지는 직4각형을 창조하고 그것을 이동하여 그 왼쪽아래구석이 창문부품자체의 왼쪽아래구석과 같아지게 한다.

QWidget::rect()함수는 창문부품자체의 자리표로 창문부품이 내접하는 직4각형을 돌려준다. (여기서 왼쪽웃구석은 0, 0이다.)

③ t10/main.cpp

```
MyWidget::MyWidget( QWidget *parent, const char *name )
  : QWidget( parent, name )
{
```

구성자는 기본적으로 같지만 일부 새로운것이 추가되였다.

```
 LCDRange *force  = new LCDRange( this, "force" );
 force->setRange( 10, 50 );
```

힘설정에 사용하는 두번째 LCDRange를 추가한다.

```
 connect( force, SIGNAL(valueChanged(int)), cannonField, SLOT(setForce(int)) );
 connect( cannonField, SIGNAL(forceChanged(int)), force, SLOT(setValue(int)) );
```

angle창문부품에 대하여 수행한것처럼 force창문부품과 cannonField창문부품을 련결한다.

```
 QVBoxLayout *leftBox = new QVBoxLayout;
 grid->addLayout( leftBox, 1, 0 );
 leftBox->addWidget( angle );
 leftBox->addWidget( force );
```

소절 9에서 배치관리자의 왼쪽아래세포에 angle을 놓는다. 이제 그 세포에 두개 창문부품을 넣으려고 하므로 수직칸을 만들고 살창세포에 수직칸을 넣고 angle과 range를 수직칸에 넣는다.

```
 force->setValue( 25 );
```

힘값을 25로 초기화한다.

(2) 동작

깜빡거림은 없어지고 힘조종요소를 가지게 되였다.

(3) 련습

포신의 크기가 힘에 의존하게 하시오.

오른쪽아래구석에 포를 놓는다.

더 좋은 건반대면부를 추가하시오. 실례로 +와 -로 힘을 증가감소시키고 사격하도록 하시오.

암시: QAccel과 LCDRange에서 QSlider::addStep()와 같은 새로운 addStep()와 subtractStep()처리부들, 왼건과 오른건의 동작(I와 !)도 변경하시오.

11. 사격하기

이 실례에서는 동화적인 사격을 실현하기 위한 시계를 소개한다.

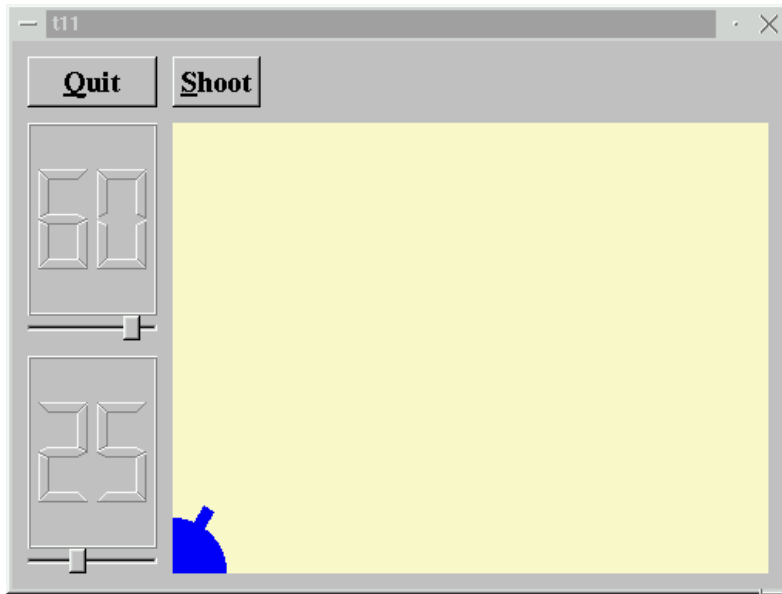



그림 2-11. 사격

- t11/lcdrange.h는 LCDRange클라스를 정의한다.
- t11/lcdrange.cpp는 LCDRange실현을 포함한다.
- t11/cannon.h는 CannonField클라스를 정의한다.
- t11/cannon.cpp는 CannonField실현을 포함한다.
- t11/main.cpp는 MyWidget와 main을 포함한다.

(1) 명령문설명

① t11/cannon.h

현재 CannonField는 사격능력을 가지고있다.

```
void   shoot();
```

이 처리부호출은 포탄이 공중에 없으면 포를 쏘게 한다.

```
private slots:
  void   moveShot();
```

이 비공개처리부는 QTimer를 사용하여 포탄이 공중에 있는동안 포탄을 이동하는데 쓰인다.

```
private:
  void   paintShot( QPainter * );
```

이 비공개함수는 포탄을 그린다.

```
QRect shotRect() const;
```

이 비공개함수는 포탄이 공중에 있으면 포탄이 내접하는 직4각형을 돌려주며 그렇지 않으면 직4각형은 정의되지 않는다.

```
int timerCount;
QTimer * autoShootTimer;
float shoot_ang;
float shoot_f;
};
```

이 비공개변수들은 포탄을 서술하는 정보를 포함한다. timerCount는 포탄이 발사된 후부터 경과된 시간을 보관한다. shoot_ang은 사격할 때 포의 각도이고 shoot_f는 포의 힘이다.

② t11/cannon.cpp

```
#include <math.h>
```

sin()과 cos()함수들이 필요하므로 수학서고를 포함한다.

```
CannonField::CannonField( QWidget *parent, const char *name )
  : QWidget( parent, name )
{
 ang = 45;
 f = 0;
 timerCount = 0;
 autoShootTimer = new QTimer( this, "movement handler" );
 connect( autoShootTimer, SIGNAL(timeout()), this,
              SLOT(moveShot()) );
 shoot_ang = 0;
 shoot_f = 0;
 setPalette( QPalette( QColor( 250, 250, 200) ) );
}
```

새로운 비공개변수들을 초기화하고 QTimer::timeout()신호를 moveShot()처리부에 련결한다. 시계가 시간을 요구할 때마다 포탄을 이동한다.

```
void CannonField::shoot()
{
 if ( autoShootTimer->isActive() )
  return;
 timerCount = 0;
 shoot_ang = ang;
 shoot_f = f;
 autoShootTimer->start( 50 );
}
```

이 함수는 공중에 포탄이 없으면 포탄을 발사한다. timerCount는 0으로 재설정된다. shoot_ang과 shoot_f는 현재 포의 각도와 힘으로 설정된다. 끝으로 시계를 기동한다.

```
void CannonField::moveShot()
{
 QRegion r( shotRect() );
 timerCount++;

 QRect shotR = shotRect();

 if ( shotR.x() > width() || shotR.y() > height() )
  autoShootTimer->stop();
 else
  r = r.unite( QRegion( shotR ) );
 repaint( r );
}
```

moveShot()는 포탄을 이동하는 처리부로서 QTimer가 발화되는 50ms간격으로

호출된다.
이 처리부는 새 위치를 계산하고 새 위치에 포탄이 있는 화면을 다시 그리고 필요하다면 시계를 중지한다.
우선 낡은 shotRect()를 보유하는 QRegion을 만든다. QRegion은 어떤 부류의 령역이나 보관할수 있으며 여기서는 그것을 사용하여 그리기를 단순화한다. shotRect()는 현재 포탄이 내접하는 직4각형을 돌려준다.
그다음 timerCount를 증가시키는데 이것은 포탄을 그 궤도를 따라 한걸음 이동하는 효과를 가진다.
다음에 새로운 포탄직4각형을 계산한다.
포탄이 창문부품의 오른변이나 밑변을 지나가면 시계를 중지하거나 새로운 shotRect()를 QRegion에 추가한다.
끝으로 QRegion을 다시 그린다. 이것은 갱신을 요구하는 하나이상의 직4각형들에 하나의 그리기사건을 보낸다.

```
void CannonField::paintEvent( QPaintEvent *e )
{
 QRect updateR = e->rect();
 QPainter p( this );

 if ( updateR.intersects( cannonRect() ) )
  paintCannon( &p );
 if ( autoShootTimer->isActive() &&
   updateR.intersects( shotRect() ) )
  paintShot( &p );
}
```

그리기사건함수는 앞항에서부터 둘로 분리하였다. 이제 그리기가 요구되는 령역의 경계직4각형을 추출하고 그것이 포 또는 필요하다면 포탄과 교차하는가 검사하고 paintCannon() 혹은 paintShot()를 호출한다.

```
void CannonField::paintShot( QPainter *p )
{
 p->setBrush( black );
 p->setPen( NoPen );
 p->drawRect( shotRect() );
}
```

이 비공개함수는 검은색의 직4각형으로 포탄을 그린다.
paintCannon()의 실현은 앞의 소절에서 paintEvent()와 같다.

```
QRect CannonField::shotRect() const
{
 const double gravity = 4;

 double time     = timerCount / 4.0;
 double velocity  = shoot_f;
 double radians   = shoot_ang*3.14159265/180;

 double velx    = velocity*cos( radians );
 double vely    = velocity*sin( radians );
 double x0  = ( barrelRect.right()  + 5 )*cos(radians);
```

```
  double y0  = ( barrelRect.right()  + 5 )*sin(radians);
  double x   = x0 + velx*time;
  double y   = y0 + vely*time - 0.5*gravity*time*time;

  QRect r = QRect( 0, 0, 6, 6 );
  r.moveCenter( QPoint( qRound(x), height() - 1 - qRound(y) ) );
  return r;
}
```

이 비공개함수는 포탄의 중심점을 계산하고 포탄이 내접하는 직4각형을 돌려준다. 이것은 초기의 포힘과 각도, 그리고 시간이 지남에 따라 증가하는 timerCount를 사용한다.

사용한 식은 중력마당에서 쓸림없는 운동에 관한 전형적인 뉴톤의 공식이다. 단순성을 위하여 아인슈타인효과를 무시하도록 선택한다.

y자리표가 우로 증가하는 자리표계에서 중점을 계산한다. 중점을 계산한 다음 6×6의 크기로 QRect를 구성하고 그 중점을 우에서 계산한 점으로 이동한다. 같은 조작으로 그 점을 창문부품의 자리표계로 변환한다(5장 3절 참고).

qRound()함수는 qglobal.h에서 정의된 inline함수이다. (다른 모든 Qt머리부파일들에 의해 포함된다.) qRound()는 double을 제일 가까운 옹근수로 둥그리기한다.

③ t11/main.cpp

```
class MyWidget: public QWidget
{
public:
  MyWidget( QWidget *parent=0, const char *name=0 );
};
```

유일한 추가는 Shoot단추이다.

```
  QPushButton *shoot = new QPushButton( "&Shoot", this, "shoot" );
  shoot->setFont( QFont( "Times", 18, QFont::Bold ) );
```

구성자에서는 Quit단추에서와 꼭같이 Shoot단추를 창조하고 설정한다. 구성자의 첫인수는 단추본문이고 셋째 인수가 창문부품의 이름이다.

```
  connect( shoot, SIGNAL(clicked()), cannonField, SLOT(shoot()) );
```

Shoot단추의 clicked()신호를 cannonField의 shoot()처리부에 련결한다.

(2) 동작

포를 사격할수 있으나 사격할것이 없다.

(3) 련습

포탄을 색있는 원으로 만드시오. 암시: QPainter::drawEllipse()를 리용할수 있다.

포탄이 공중에 있을 때 포의 색을 변경하시오.

12. 공중에 목표를 띄우기

이 실례에서는 LCDRange클라스를 확장하여 본문표식자를 포함한다. 또한 사격동작을 제공한다.

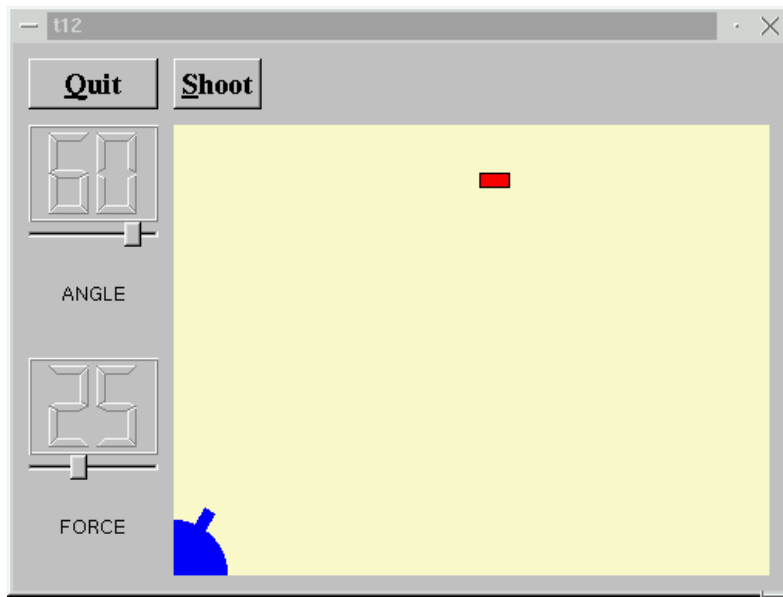



그림 2-12. 목표를 띄우기

- t12/lcdrange.h는 LCDRange클라스를 정의한다.
- t12/lcdrange.cpp는 LCDRange실현을 포함한다.
- t12/cannon.h는 CannonField클라스를 정의한다.
- t12/cannon.cpp는 CannonField를 실현한다.
- t12/main.cpp는 MyWidget와 main을 포함한다.

(1) 명령문설명

① t12/lcdrange.h

현재 LCDRange는 본문표식자를 가지고있다.

```
class QLabel;
```

클라스정의에서 QLabel의 지적자를 사용하려고 하므로 그 클라스를 선언한다.

```
class LCDRange : public QVBox
{
  Q_OBJECT
public:
  LCDRange( QWidget *parent=0, const char *name=0 );
  LCDRange( const char *s, QWidget *parent=0, const char *name=0 );
```

parent, name과 함께 표식자본문을 설정하는 새로운 구성자를 추가하였다.

```
  const char *text()  const;
```

이 함수는 표식자본문을 돌려준다.

```
  void setText( const char * );
```

이 처리부는 표식자본문을 설정한다.

```
private:
  void init();
```

현재 두개의 구성자를 가지므로 비공개 init()함수에 공동초기화를 넣기로 선택하였다.

```
  QLabel   *label;
```

또한 새로운 비공개변수 즉 QLabel을 가진다. QLabel은 Qt의 표준창문부품들중 하나이고 틀이 있거나 없는 본문이나 픽스매프를 표시할수 있다.

② t12/lcdrange.cpp

```
#include <qlabel.h>
```

여기서 QLabel클라스를 정의한다.

```
LCDRange::LCDRange( QWidget *parent, const char *name ) : QVBox( parent, name )
{
 init();
}
```

이 구성자는 일반초기화코드를 포함하는 init()함수를 호출한다.

```
LCDRange::LCDRange( const char *s, QWidget *parent, const char *name )
    : QVBox( parent, name )
{
 init();
 setText( s );
}
```

이 구성자는 우선 init()를 호출하고 표식자본문을 설정한다.

```
void LCDRange::init()
{
 QLCDNumber *lcd  = new QLCDNumber( 2, this, "lcd"  );
 slider = new QSlider( Horizontal, this, "slider" );
 slider->setRange( 0, 99 );
 slider->setValue( 0 );
 label = new QLabel( " ", this, "label"  );
 label->setAlignment( AlignCenter );
 connect( slider, SIGNAL(valueChanged(int)),
    lcd, SLOT(display(int)) );
 connect( slider, SIGNAL(valueChanged(int)),
    SIGNAL(valueChanged(int)) );

 setFocusProxy( slider );
}
```

lcd와 slider의 설정은 앞항에서와 같다. 다음에 QLabel을 창조하고 내용을 수직과 수평으로 중심에 배치한다. connect()문은 또한 앞의 소절로부터 취하였다.

```
const char *LCDRange::text() const
{
 return label->text();
}
```

이 함수는 표식자본문을 돌려준다.

```
void LCDRange::setText( const char *s )
{
 label->setText( s );
}
```

이 함수는 표식자본문을 설정한다.

③ t12/cannon.h

현재 CannonField는 두개의 신호 hit()와 missed()를 가지며 목표를 포함한다.

```
void newTarget();
```

이 처리부는 새 위치에 목표를 창조한다.

```
signals:
  void hit();
  void missed();
```

hit()신호는 포탄이 목표를 맞힐 때 발생된다. missed()신호는 포탄이 창문부품의 오른변이나 밑변을 지나갈 때 발생된다.

```
void paintTarget( QPainter * );
```

이 비공개함수는 목표를 그린다.

```
QRect targetRect() const;
```

이 비공개함수는 목표가 내접하는 직4각형을 돌려준다.

```
QPoint target;
```

이 비공개변수는 목표의 중점을 포함한다.

④ t12/cannon.cpp

```
#include <qdatetime.h>
```

QDate, QTime 그리고 QDateTime클라스정의들을 포함한다.

```
#include <stdlib.h>
```

rand()함수가 요구되므로 stdlib서고를 포함한다.

```
newTarget();
```

이 행을 구성자에 추가하여 목표의 우연적인 위치를 창조한다. 사실 newTarget()함수는 목표를 그리려고 한다. 구성자에 있으므로 CannonField창문부품은 보이지 않는다. Qt는 숨겨진 창문부품에 대하여 repaint()를 호출할 때 불필요한 작업이 진행되지 않도록 한다.

```
void CannonField::newTarget()
{
  static bool first_time = TRUE;
  if ( first_time ) {
   first_time = FALSE;
   QTime midnight( 0, 0, 0 );
   srand( midnight.secsTo(QTime::currentTime()) );
  }
  QRegion r( targetRect() );
  target = QPoint( 200 + rand() % 190, 10  + rand() % 255 );
  repaint( r.unite( targetRect() ) );
}
```

이 비공개함수는 새로운 우연위치에 목표중점을 창조한다.

rand()함수에 의해 우연옹근수를 생성한다. 보통 rand()함수는 프로그람을 실행할 때마다 같은 계렬의 수를 돌려준다. 이것은 목표가 매번 같은 위치에 나타나게 한다. 이것을 피하기 위하여 함수가 처음으로 호출될 때 우연수종자(seed)를 설정해야 한다. 우연수종자도 역시 같은 우연수계렬을 피할수 있도록 우연적이여야 하므로 거짓우연수값으로서 자정으로부터 경과한 초수를 사용하는것이다.

우선 정적인 bool국부변수를 창조하여 함수호출들사이에서 값을 보관하도록 한다.

if시험은 if블로크안에서 first_time을 FALSE로 설정하였으므로 newTarget()함수가 처음으로 호출될 때만 성공한다.

그다음 QTime객체 midnight를 창조하고 시간 00:00:00을 표시한다. 다음에 자정으로부터 현재까지의 초수를 계산하여 우연수종자로 사용한다(QDate, QTime,

QDateTime 참고).

끝으로 목표의 중점을 계산한다. 중점을 직4각형(x=200, y=35, width=190, height=255)안에서 유지하고(가능한 x와 y값은 x = 200～389, y = 35～289) 자리표계에서 y위치를 창문부품의 밑변에서 0으로 하고 우로 가면서 증가하게 하며 x는 보통과 같이 왼변을 0으로 하고 오른쪽으로 가면서 증가하게 한다.

실험에 의하면 이것이 늘 포탄의 도달범위에 있다는것을 알수 있다.

rand()는 0이상의 우연옹근수를 돌려준다.

```
void CannonField::moveShot()
{
  QRegion r( shotRect() );
  timerCount++;

  QRect shotR = shotRect();
```

시계사건의 이 부분은 변경하지 않았다.

```
  if ( shotR.intersects( targetRect() ) ) {
    autoShootTimer->stop();
    emit hit();
```

if문은 포탄직4각형이 목표직4각형과 교차하는가 검사한다. 교차하면 포탄은 목표를 맞힌다. 이때 사격시계를 정지하고 hit()신호를 발생하여 외부세계에 목표가 파괴되였다고 알리고 되돌아간다.

현지에 새 목표를 창조할수 있으나 CannonField이 부분품이므로 이것은 사용자에게 맡긴다.

```
  } else if ( shotR.x() > width() || shotR.y() > height() ) {
    autoShootTimer->stop();
    emit missed();
```

이 if문은 앞의 소절과 같지만 missed()신호를 발생하여 외부세계에 실패를 알린다는것만 다르다.

```
  } else {
```

그리고 함수의 나머지부분은 이전과 같다.

CannonField::paintEvent()는 이전과 같지만 다음것을 추가한다.

```
  if ( updateR.intersects( targetRect() ) )
    paintTarget( &p );
```

우의 두행은 필요할 때 목표를 그린다.

```
void CannonField::paintTarget( QPainter *p )
{
  p->setBrush( red );
  p->setPen( black );
  p->drawRect( targetRect() );
}
```

이 비공개함수는 목표를 적색으로 칠하고 흑색륜곽선을 그린다.

```
QRect CannonField::targetRect() const
{
  QRect r( 0, 0, 20, 10 );
  r.moveCenter( QPoint(target.x(),height() - 1 - target.y()) );
  return r;
}
```

이 비공개함수는 목표가 내접하는 직4각형을 돌려준다. newTarget()에서 target점은 창문부품의 밑변을 y자리표 0으로 사용한다. QRect::moveCenter()를 호출하기전에 창문부품자리표로 점을 계산한다.

이 자리표변환을 선택한 리유는 목표와 창문부품밑변사이의 거리를 고정하려는데 있다. 창문부품의 크기는 임의의 시간에 사용자가 프로그람에 의해 조절할수 있다.

⑤ t12/main.cpp

MyWidget클라스에 새로운 성원은 없지만 구성자를 변경하여 새로운 LCDRange 본문표식자를 설정한다.

```
LCDRange *angle  = new LCDRange( "ANGLE", this, "angle" );
```

angle본문표식자를 "ANGLE"로 설정한다.

```
LCDRange *force  = new LCDRange( "FORCE", this, "force" );
```

force본문표식자를 "FORCE"로 설정한다.

(2) 동작

LCDRange창문부품들은 좀 이상해보인다. QVBox의 내부배치관리는 표식자들에 더 많은 공간을 주며 나머지가 충분하지 않다. 다음 소절에서 그것을 고착시킨다.

(3) 련습

누를 때 포탄궤도를 5s동안 현시하는 거짓(cheat)단추를CannonField에 만드시오.

앞의 소절에서 《둥근포탄》련습을 하였다면 shotRect()를 QRegion을 돌려주는 shotRegion()으로 변경하여 실제로 정확한 충돌탐색을 하도록 하시오.

이동하는 목표를 만드시오.

목표가 늘 화면우에 완전히 창조되는가 확신하시오.

창문부품크기를 조절할수 없으므로 목표를 볼수 없다는것을 확인하시오. 암시: QWidget::setMinimumSize()를 사용하시오.

간단하지 않지만 공중에 여러개의 포탄이 동시에 있게 하시오.(암시:Shot객체를 만드시오.)

13. 유희의 완료

이 실례에서는 득점을 가지고 실제로 놀수 있는 유희로 만든다. MyWidget에 새로운 이름(GameBoard)을 주고 일부 처리부들을 추가한다.

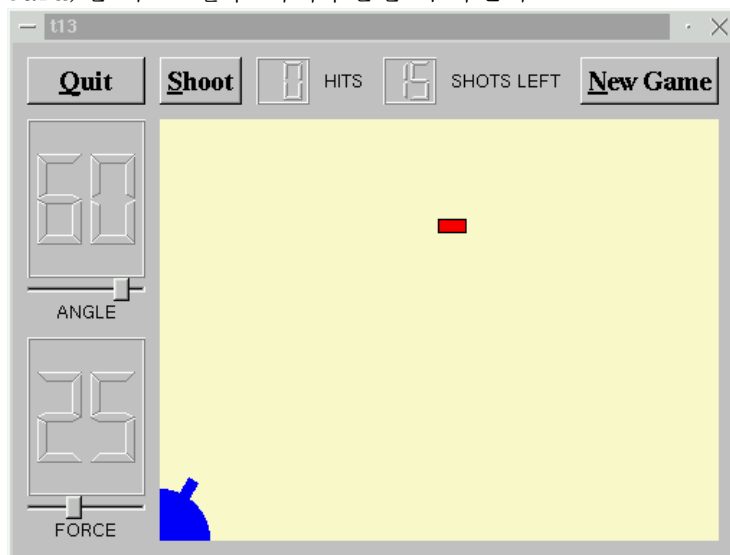



그림 2-13. 유희의 완료

gamebrd.h에 정의를, gamebrd.cpp에 실현을 보관한다.
그러면 CannonField가 유희완료상태를 가진다.
LCDRange에서 배치문제들이 고착된다.

- t13/lcdrange.h는 LCDRange클라스를 정의한다.
- t13/lcdrange.cpp는 LCDRange실현을 포함한다.
- t13/cannon.h는 CannonField클라스를 정의한다.
- t13/cannon.cpp는 CannonField실현을 포함한다.
- t13/gamebrd.h는 GameBoard클라스를 정의한다.
- t13/gamebrd.cpp는 GameBoard실현을 포함한다.
- t13/main.cpp는 MyWidget와 main을 포함한다.

(1) 명령문설명

① t13/lcdrange.h

```
#include <qwidget.h>

class QSlider;
class QLabel;

class LCDRange : public QWidget
```

QVBox보다도 QWidget를 계승한다. QVBox는 아주 사용하기 쉽지만 더 강력하고 복잡한 QVBoxLayout로 절환된다. (QVBoxLayout는 창문부품이 아니고 창문부품을 관리한다.)

② t13/lcdrange.cpp

```
#include <qlayout.h>
```

이제 qlayout.h를 포함하여 다른 배치관리API를 얻을수 있다.

```
LCDRange::LCDRange( QWidget *parent, const char *name ):QWidget( parent, name )
```

보통의 방법으로 QWidget를 계승한다.
다른 구성자는 같은 방법으로 변경한다. init()는 변경되지 않지만 끝에 일부 행을 추가한다.

```
QVBoxLayout * l = new QVBoxLayout( this );
```

모두 기정값을 가지는 QVBoxLayout를 창조하고 이 창문부품의 자식들을 관리한다.

```
l->addWidget( lcd, 1 );
```

짝대기에 신축비가 0아닌 QLCDNumber를 추가한다.

```
l->addWidget( slider );
l->addWidget( label );
```

그다음 기정신축비 0을 가지는 두개 창문부품을 추가한다.
이 신축조종은 QVBoxLayout와 QHBoxLayout 및 QGridLayout)에서 제공되고 QVBox와 같은 클라스에서는 제공되지 않는다. 이 경우에 QLCDNumber가 신축되고 다른것들은 신축되지 말아야 한다.

③ t13/cannon.h

이제는 CannonField가 유희완료상태와 일부 새로운 함수들을 가진다.

```
bool  gameOver() const { return gameEnded; }
```

이 함수는 유희가 완료하면 TRUE, 유희가 진행되고있으면 FALSE를 돌려준다.

```
void  setGameOver();
void  restartGame();
```

여기에 두개의 새로운 처리부 setGameOver()와 restartGame()이 있다.

```
void  canShoot( bool );
```

새 신호들은 CannonField가 shoot()처리부가 의미를 가지는 상태에 있다는것을 가리킨다. 다음에 그것을 사용하여 Shoot단추를 허용, 금지한다.

```
bool gameEnded;
```

이 비공개변수는 유희의 상태를 포함한다. TRUE는 유희가 끝났다는것을 의미하고 FALSE는 유희가 진행되고있다는것을 의미한다.

④ t13/cannon.cpp

```
gameEnded = FALSE;
```

이 행을 구성자에 추가하였다. 처음에 유희는 끝나지 않는다.

```
void CannonField::shoot()
{
 if ( isShooting() )
  return;
 timerCount = 0;
 shoot_ang = ang;
 shoot_f = f;
 autoShootTimer->start( 50 );
 emit canShoot( FALSE );
}
```

새로운 isShooting()함수를 추가함으로써 shoot()가 그것을 직접시험대신에 사용하게 한다. 또한 shoot는 CannonField가 지금 사격할수 없다는것을 알린다.

```
void CannonField::setGameOver()
{
 if ( gameEnded )
  return;
 if ( isShooting() )
  autoShootTimer->stop();
 gameEnded = TRUE;
 repaint();
}
```

이 처리부는 유희를 끝낸다. 이 창문부품이 유희완료시각을 모르므로 외부의 CannonField로부터 호출되여야 한다. 이것은 부분품프로그람작성에서 중요한 설계원칙이다. 부분품을 될수록 유연하게 하여 각이한 규칙들에서 사용할수 있게 한다. (실례로 여러선수 유희에서는 변경하지 않은 CannonField를 사용할수 있다.)

유희가 이미 끝났으면 즉시 되돌아간다. 유희가 진행되고있으면 포탄을 세우고 유희완료기발을 설정하고 전체 위체트를 다시 그린다.

```
void CannonField::restartGame()
{
 if ( isShooting() )
  autoShootTimer->stop();
 gameEnded = FALSE;
 repaint();
 emit canShoot( TRUE );
}
```

이 처리부는 새로운 유희를 시작한다. 포탄이 공중에 있으면 사격을 중지한 다음

gameEnded 변수를 재설정하고 창문부품을 다시 그린다.

moveShot()도 역시 hit() 혹은 miss()와 같은 시간에 새로운 canShoot(TRUE) 신호를 발생한다.

CannonField::paintEvent()에서의 수정

```
void CannonField::paintEvent( QPaintEvent *e )
{
 QRect updateR = e->rect();
 QPainter p( this );

 if ( gameEnded ) {
  p.setPen( black );
  p.setFont( QFont( "Courier", 48, QFont::Bold ) );
  p.drawText( rect(), AlignCenter, "Game Over" );
 }
```

그리기사건은 유희가 끌나면 즉 gameEnded가 TRUE이면 본문 "Game Over"를 현시하도록 확장되였다. 유희가 끌날 때 속도는 중요하지 않다.

우선 본문을 그리기 위하여 검은펜을 설정한다. 펜색은 본문을 그릴 때 사용된다. 다음에 Courier계렬에서 48point강조서체를 선택한다. 끝으로 창문부품의 직4각형중심에 본문을 그린다. 일부 체계(특히 유니코드서체를 가지는 X봉사기들)에서 큰 서체를 적재할수 있다. Qt가 서체를 보관하므로 서체가 처음으로 사용될 때 이것을 통지한다.

```
 if ( updateR.intersects( cannonRect() ) )
  paintCannon( &p );
 if ( isShooting() && updateR.intersects( shotRect() ) )
  paintShot( &p );
 if ( !gameEnded && updateR.intersects( targetRect() ) )
  paintTarget( &p );
}
```

사격할 때만 포탄을 그리고 유희할 때(즉 유희가 끝나지 않았을 때)만 목표를 그린다.

⑤ t13/gamebrd.h

이 파일은 새로운것이다. 이것은 GameBoard클라스의 정의를 포함한다.

```
class QPushButton;
class LCDRange;
class QLCDNumber;
class CannonField;

#include "lcdrange.h"
#include "cannon.h"

class GameBoard : public QWidget
{
 Q_OBJECT
public:
 GameBoard( QWidget *parent=0, const char *name=0 );

protected slots:
 void  fire();
```

```
    void  hit();
    void  missed();
    void  newGame();

   private:
    QLCDNumber  *hits;
    QLCDNumber  *shotsLeft;
    CannonField *cannonField;
   };
```

현재 4개 처리부를 추가하였다. 이것들은 보호되고 내적으로 사용된다. 또한 유희상태를 현시하는 두개의 QLCDNumber (hits와 shotsLeft)를 추가하였다.

⑥ t13/gamebrd.cpp

이 파일은 새로운것이다. 이것은 마감에 GameBoard클라스의 실현을 포함한다.

GameBoard구성자을 일부 변경한다.

```
cannonField = new CannonField( this, "cannonField" );
```

지금 cannonField는 성원변수이므로 그것을 사용하도록 구성자를 조심히 변경한다.

```
connect( cannonField, SIGNAL(hit()), this, SLOT(hit()) );
connect( cannonField, SIGNAL(missed()), this, SLOT(missed()) );
```

이번에는 처리부가 목표를 맞히거나 놓칠 때 어떤 일을 하기 위하여 CannonField의 hit()와 missed()신호를 이 클라스에서와 같은 이름을 가지는 두개의 보호처리부와 련결한다.

```
connect( shoot, SIGNAL(clicked()), SLOT(fire()) );
```

이미 Shoot단추의 clicked()신호를 직접 CannonField의 shoot()처리부에 련결하였다. 이번에는 발사한 포탄수를 보관하여 이 클라스의 보호처리부에 련결하려고 한다.

자체에 포함한 콤포넨트들과 작업할 때 프로그람의 동작을 변경하는것은 아주 간단하다.

```
connect( cannonField, SIGNAL(canShoot(bool)), shoot, SLOT(setEnabled(bool)) );
```

또한 cannonField의 canShoot()신호를 리용하여 Shoot단추를 적당히 허용하거나 금지한다.

```
QPushButton *restart = new QPushButton( "&New Game", this, "newgame" );
restart->setFont( QFont( "Times", 18, QFont::Bold ) );

connect( restart, SIGNAL(clicked()), this, SLOT(newGame()) );
```

다른 단추들에서 수행한것처럼 New Game단추를 창조하고 설정하고 련결한다. 이 단추를 찰칵하면 이 창문부품에서 newGame()처리부가 동작한다.

```
hits = new QLCDNumber( 2, this, "hits" );
shotsLeft = new QLCDNumber( 2, this, "shotsleft" );
QLabel *hitsL = new QLabel( "HITS", this, "hitsLabel" );
QLabel *shotsLeftL= new QLabel( "SHOTS LEFT", this, "shotsleftLabel" );
```

4개의 새로운 창문부품을 창조한다. GameBoard클라스에서 QLabel창문부품들의 지적자를 사용할 필요가 없으므로 그것을 보관하지 않는다. Qt는 GameBoard창문부품이 해체될 때 그것들을 삭제하며 배치관리자클라스들은 그것들의 크기를 적당히 조절한다.

```
QHBoxLayout *topBox = new QHBoxLayout;
```

```
grid->addLayout( topBox, 0, 1 );
topBox->addWidget( shoot );
topBox->addWidget( hits );
topBox->addWidget( hitsL );
topBox->addWidget( shotsLeft );
topBox->addWidget( shotsLeftL );
topBox->addStretch( 1 );
topBox->addWidget( restart );
```

오른쪽 우에 있는 세포안의 창문부품수는 많다. 그것이 비였으면 현재는 배치설정을 모두 묶어서 더 잘 개괄하는것이 좋다.

New Game단추의 왼쪽에 범위를 넣을 대신에 모든 창문부품들이 자기가 좋아하는 크기를 가지게 한다.

```
newGame();
}
```

GameBoard를 모두 구성하였으므로 newGame()에 의하여 그것을 기동한다. (NewGame()은 처리부이지만 일반함수처럼 사용될수도 있다.)

```
void GameBoard::fire()
{
 if ( cannonField->gameOver() || cannonField->isShooting() )
  return;
 shotsLeft->display( shotsLeft->intValue() - 1 );
 cannonField->shoot();
}
```

이 함수는 포탄을 발사한다. 유희가 끝나거나 포탄이 공중에 있으면 즉시 되돌아간다. 포탄수를 하나 감소하고 포를 발사하도록 한다.

```
void GameBoard::hit()
{
 hits->display( hits->intValue() + 1 );
 if ( shotsLeft->intValue() == 0 )
  cannonField->setGameOver();
 else
  cannonField->newTarget();
}
```

이 처리부는 포탄이 목표를 맞힐 때 동작하며 명중회수를 증가시킨다. 포탄이 없으면 유희는 끝나고 그렇지 않으면 CannonField이 새로운 목표를 생성한다.

```
void GameBoard::missed()
{
 if ( shotsLeft->intValue() == 0 )
  cannonField->setGameOver();
}
```

이 처리부는 포탄이 목표를 놓쳤을 때 동작한다. 포탄이 없으면 유희는 끝난다.

```
void GameBoard::newGame()
{
 shotsLeft->display( 15 );
 hits->display( 0 );
 cannonField->restartGame();
```

```
cannonField->newTarget();
}
```

이 처리부는 사용자가 Restart단추를 찰칵할 때 동작한다 . 또한 구성자로부터도 호출된다. 우선 포탄수를 15로 설정한다. 이것은 포탄수를 설정하는 유일한 위치로서 유희규칙을 변경하려고 할 때 변경한다. 다음에 명중회수를 재설정하고 유희를 다시 시작하고 새 목표를 생성한다.

⑦ t13/main.cpp

창문부품을 작성하였으며 이제 유일하게 남은것은 main()함수이다.

(2) 동작

포는 목표에 발사할수 있고 목표를 맞힐 때 새 목표가 자동적으로 창조된다.

명중회수와 나머지 포탄이 현시되고 프로그람은 그것을 보관한다. 유희를 끝내고 새 유희를 시작하는 단추가 있다.

(3) 련습

우연적인 바람결수를 추가하고 그것을 사용자에게 보여주시오.

포탄이 목표를 맞힐 때 산탄효과를 만드시오.

여러 목표를 실현하시오.

14. 방탄벽을 쌓기

이것은 마지막 실례 즉 완성된 유희이다.

건반지름건을 추가하고 CannonField에 마우스사건들을 받아들인다. CannonField주위에 틀을 넣고 방탄벽을 추가하여 유희를 더 재미있게 만든다.

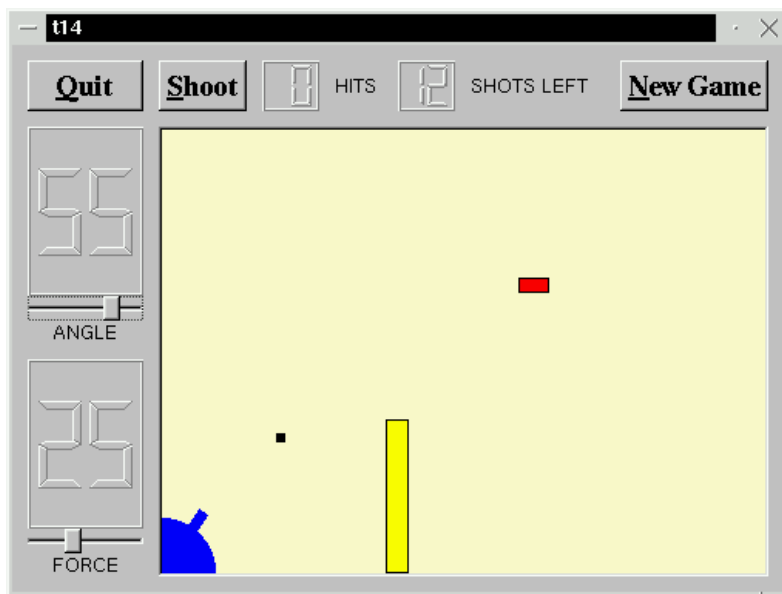



그림 2-14. 방탄벽쌓기

- t14/lcdrange.h는 LCDRange클라스를 정의한다.
- t14/lcdrange.cpp는 LCDRange를 실현한다.
- t14/cannon.h는 CannonField클라스를 정의한다.
- t14/cannon.cpp는 CannonField를 실현한다.
- t14/gamebrd.h는 GameBoard클라스를 정의한다.
- t14/gamebrd.cpp는 GameBoard를 실현한다.

• t14/main.cpp는 MyWidget와 main을 포함한다.

(1) 명령문설명

① t14/cannon.h

지금 CannonField는 마우스사건들을 받아들이고 사용자가 방탄벽우를 찰칵하고 끌어서 방탄벽을 겨눌수 있다. 또한 CannonField는 방탄벽을 가진다.

```
protected:
 void  paintEvent( QPaintEvent * );
 void  mousePressEvent( QMouseEvent * );
 void  mouseMoveEvent( QMouseEvent * );
 void  mouseReleaseEvent( QMouseEvent * );
```

이미 친숙해진 사건처리함수들과 함께 CannonField는 마우스사건처리함수들을 실현한다.

```
 void  paintBarrier( QPainter * );
```

이 비공개함수는 방탄벽을 그린다.

```
 QRect barrierRect() const;
```

이 비공개함수는 방탄벽이 내접하는 직4각형을 돌려준다.

```
 bool  barrelHit( const QPoint & ) const;
```

이 비공개함수는 점이 포의 방탄벽안에 있는가 검사한다.

```
 bool barrelPressed;
```

이 비공개변수는 사용자가 방탄벽우에서 마우스를 누른 상태에서 놓지 않았을 때 TRUE이다.

② t14/cannon.cpp

```
 barrelPressed = FALSE;
```

이 행을 구성자에 추가하였다. 처음에 마우스는 방탄벽우에서 눌리우지 않는다.

```
 } else if ( shotR.x() > width() || shotR.y() > height() ||
    shotR.intersects(barrierRect()) ) {
```

이제는 방탄벽이 있지만 세가지 빠뜨린것이 있다. 세번째로 시험한다.

```
void CannonField::mousePressEvent( QMouseEvent *e )
{
 if ( e->button() != LeftButton )
  return;
 if ( barrelHit( e->pos() ) )
  barrelPressed = TRUE;
}
```

이것은 Qt사건처리함수이다. 이것은 마우스유표가 창문부품우에 있고 사용자가 마우스단추를 누르는 경우에 호출된다.

사건이 왼쪽마우스단추에 의해 생성되지 않았으면 즉시 되돌아간다. 그렇지 않으면 마우스유표의 위치가 포의 방탄벽안에 있는가 검사하고 있으면 barrelPressed를 TRUE로 설정한다.

pos()함수는 창문부품자리표계의 점을 돌려준다.

```
void CannonField::mouseMoveEvent( QMouseEvent *e )
{
 if ( !barrelPressed )
  return;
 QPoint pnt = e->pos();
```

```
  if ( pnt.x() <= 0 )
   pnt.setX( 1 );
  if ( pnt.y() >= height() )
   pnt.setY( height() - 1 );
  double rad = atan(((double)rect().bottom()-pnt.y())/pnt.x());
  setAngle( qRound ( rad*180/3.14159265 ) );
 }
```

이것은 또 하나의 Qt사건처리함수로서 사용자가 창문부품안에서 이미 마우스단추를 누르고 마우스를 이동하거나 끌기했을 때 호출된다. (단추들을 누르지 않을 때도 Qt가 마우스이동사건들을 보내게 할수 있다. QWidget::setMouseTracking() 참고.)

이 처리함수는 마우스유표의 위치에 따라서 포의 방탄벽을 재배치한다.

우선 방탄벽을 누르지 않으면 되돌아간다. 다음으로 마우스유표의 위치를 계산한다. 마우스유표가 창문부품의 왼쪽이나 아래에 있으면 점이 창문부품안에 놓이도록 조절한다.

그다음 창문부품의 왼쪽아래구석과 유표위치사이의 가상선과 창문부품밑변사이의 각을 계산한다. 끝으로 포의 각도를 도로 변환한 새 값으로 설정한다.

setAngle()는 포를 다시 그린다.

```
 void CannonField::mouseReleaseEvent( QMouseEvent *e )
 {
  if ( e->button() == LeftButton )
   barrelPressed = FALSE;
 }
```

이 Qt사건처리함수는 사용자가 창문부품안에서 눌렀던 마우스단추를 놓을 때마다 호출된다.

왼쪽단추를 놓으면 방탄벽을 더는 누르지 않는다는것을 확인할수 있다.

그리기사건은 두개의 행을 추가적으로 가진다. 즉

```
  if ( updateR.intersects( barrierRect() ) )
   paintBarrier( &p );
```

paintBarrier()는 paintShot(), paintTarget(), paintCannon()와 같은 일을 수행한다.

```
 void CannonField::paintBarrier( QPainter *p )
 {
  p->setBrush( yellow );
  p->setPen( black );
  p->drawRect( barrierRect() );
 }
```

이 비공개함수는 방탄벽을 황색으로 도색하고 검은색륜곽선을 가지는 직4각형으로 그린다.

```
 QRect CannonField::barrierRect() const
 {
  return QRect( 145, height() - 100, 15, 100 );
 }
```

이 비공개함수는 방탄벽의 직4각형을 돌려준다. 방탄벽의 밑변을 창문부품의 밑변으로 고정한다.

```
 bool CannonField::barrelHit( const QPoint &p ) const
 {
  QWMatrix mtx;
```

```
  mtx.translate( 0, height() - 1 );
  mtx.rotate( -ang );
  mtx = mtx.invert();
  return barrelRect.contains( mtx.map(p) );
 }
```

이 함수는 점이 방탄벽안에 있으면 TRUE, 그렇지 않으면 FALSE를 돌려준다.

여기서는 클라스 QWMatrix를 사용한다. 이것은 qpainter.h에 의하여 포함되는 머리부파일 qwmatrix.h에서 정의된다.

QWMatrix는 자리표계변환을 정의하며 QPainter와 같은 변환을 수행한다.

여기서는 paintCannon()함수에서 방탄벽을 그릴 때 수행한것과 같은 변환단계들을 수행한다. 우선 자리표계를 변환하고 그것을 회전한다.

이제는 점 p(창문부품자리표)가 방탄벽안에 놓이는가 검사하여야 한다. 그러기 위하여 변환행렬을 역전시킨다. 역행렬은 방탄벽을 그릴 때 사용하는 역변환을 수행한다. 역행렬을 리용하여 점 p를 넘기고 그것이 원시방탄벽직4각형안에 있으면 TRUE를 돌려준다.

③ t14/gamebrd.cpp

```
 #include <qaccel.h>
```

QAccel의 클라스를 정의한다.

```
  QVBox *box = new QVBox( this, "cannonFrame" );
  box->setFrameStyle( QFrame::WinPanel | QFrame::Sunken );
  cannonField = new CannonField( box, "cannonField" );
```

QVBox를 창조하고 설정하며 그 틀형식을 설정한 다음 그 칸의 자식으로서 CannonField를 창조한다. 그밖에 칸안에 아무것도 없으므로 QVBox는 CannonField주위에 틀을 넣은 효과가 나타난다.

```
  QAccel *accel = new QAccel( this );
  accel->connectItem( accel->insertItem( Key_Enter ), this,
SLOT(fire()) );
  accel->connectItem( accel->insertItem( Key_Return ), this,
SLOT(fire()) );
```

여기서는 지름건을 창조하고 설정한다. 지름건은 건반사건들을 응용프로그람에 받아들이고 주어진 건들을 누르면 처리부들을 호출하는 객체이다. 지름건은 창문부품의 자식이며 창문부품이 해체될 때 같이 해체된다. QAccel은 창문부품이 아니고 그 부모에서 효과를 볼수 없다.

두개의 지름건을 정의한다. 사용자가 Enter를 누를 때 처리부 fire()가 호출되게 하고 건 Ctrl+Q를 누를 때 응용프로그람을 중지한다. 흔히 Enter는 Return이고 두개의 건이 다 있는 건반도 있으므로 Enter와 Return이 모두 fire()를 호출하게 한다.

```
  accel->connectItem( accel->insertItem( CTRL+Key_Q ), qApp,
SLOT(quit()) );
```

그다음 Ctrl+Q를 설정하여 Alt+Q와 같은 일을 수행한다. 일부 사람들은 Ctrl+Q를 더 사용한다.

CTRL, Key_Enter, Key_Return, Key_Q는 모두 Qt에 의해 제공되는 상수들이며 실제로 Qt::Key_Enter 등과 같지만 모든 클라스들은 Qt이름공간클라스를 계승한다.

```
  QGridLayout *grid = new QGridLayout( this, 2, 2, 10 );
  grid->addWidget( quit, 0, 0 );
  grid->addWidget( box, 1, 1 );
  grid->setColStretch( 1, 10 );
```

오른쪽 아래 세포에 CannonField가 아니라 box (QVBox)를 넣는다.

(2) 동작

현재 포는 Enter를 누르면 발사된다. 또한 마우스를 사용하여 포의 사각을 지정할 수 있다. 방탄벽은 유희를 좀 더 재미있게 놀수 있게 한다. 또한 CannonField둘레에 아름다운 틀이 있다.

(3) 련습

공중비적유희를 쓰시오.

파편(Breakout)유희를 쓰시오.

제3절. 도표의 작성

이 절에서는 2절보다 현실에 더 가까운 Qt프로그람작성실례를 제시한다. 여기서는 차림표(최근파일목록 포함), 도구띠, 대화칸의 창조, 사용자환경설정의 적재와 보관 등 Qt프로그람작성의 많은수법들을 소개한다.

여기서는 사용자가 입력하는 자료에 기초하여 간단한 원형도표와 막대도표를 현시하는데 사용되는 chart라는 하나의 응용프로그람을 개발한다.

응용프로그람의 개발을 설명하고 코드부분에 대하여 설명한다. 응용프로그람의 완성된 원천은 examples/chart에 있다.

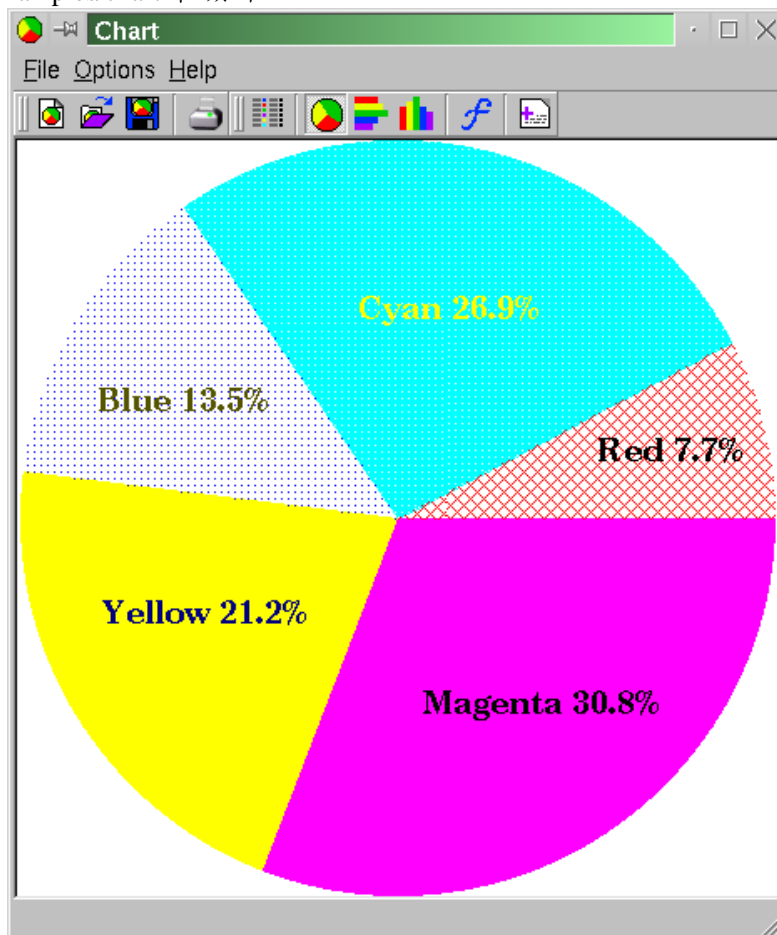



그림 2-15. 도표프로그람의 실행화면

1. 도표프로그람의 구성

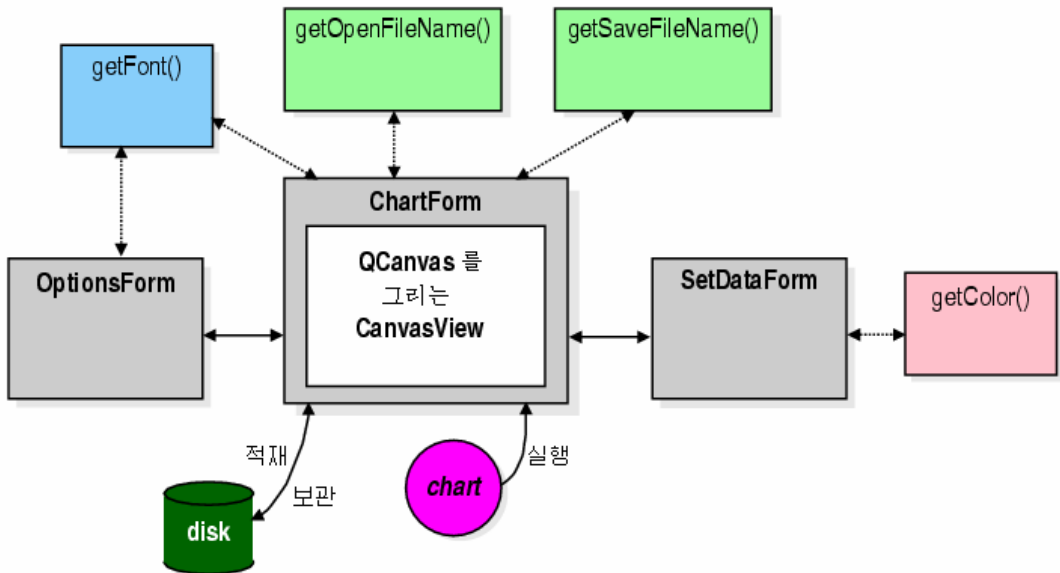



그림 2-16. 도표프로그람의 구성

chart프로그람은 사용자가 단순자료모임을 창조하고 보관하고 적재하고 표시한다. 사용자가 입력하는 매개 자료요소는 부채형이나 막대, 표식자본문, 본문의 위치와 색으로 주어질수 있다. Element클라스는 자료요소들을 재현하는데 쓰인다.

프로그람은 도표폼을 적재하는 단순한 main.cpp로 구성된다. 도표폼은 프로그람의 기능호출을 제공하는 차림표띠와 도구띠를 가진다. 프로그람은 두개의 대화칸 즉 하나는 선택대화칸, 다른 하나는 자료모임을 창조하고 편집하기 위한 대화칸을 제공한다. 두 대화칸은 도표로부터 차림표선택이나 도구띠단추를 통하여 펼쳐진다.

도표폼의 기본창문부품은 원형도표나 막대그라프를 그리는 QCanvas를 현시하는 QCanvasView이다. QCanvasView의 파생클라스를 만들어 구체적인 동작을 얻는다. 마찬가지로 표준클라스가 제공하는것보다 훨씬 더 많은것을 요구하므로 QCanvasText의 파생클라스를 만든다.(캔버스에 본문항목들을 배치하는데 쓰인다.)

프로젝트파일 chart.pro는 응용프로그람을 구축하는데 쓰이는 Makefile의 창조에 사용된다.

2. 자료요소

Element라는 C++클라스를 호출하여 자료요소들의 기억과 호출을 제공한다.

```
// element.h로부터 발취
 private:
  double m_value;
  QColor m_valueColor;
  int m_valuePattern;
  QString m_label;
  QColor m_labelColor;
  double m_propoints[2 * MAX_PROPOINTS];
```

매개 요소는 값을 가진다. 매개 값은 특정한 색과 채우기패턴으로 그라프적으로 현시된다. 값들은 그것들과 련관된 표식자를 가질수 있으며 표식자는 표식자색으로 그려지고 매개 형의 도표에 대하여 m_propoints배렬에 보관된 상대위치를 가진다.

```
#include <qcolor.h>
#include <qnamespace.h>
#include <qstring.h>
#include <qvaluevector.h>
```

Element클라스가 순수 내부적인 자료클라스라고 할지라도 4개의 Qt클라스들을 포함한다. Qt는 흔히 순수 GUI도구일식으로서 알고있지만 그밖에도 수많은 비GUI클라스들을 제공하여 응용프로그람작성의 대부분의 측면들을 유지한다. qcolor.h를 사용하여 Element클라스에 그리기색과 본문색을 보관할수 있다. qnamespace.h의 사용은 명백하다. 대부분의 Qt클라스는 각종 렬거형을 포함하는 Qt기초클라스로부터 파생된다. Element클라스는 Qt로부터 파생되지 않으므로 qnamespace.h를 포함하여 Qt렬거이름들을 호출할수 있게 한다. 다른 수법은 Element를 Qt파생클라스로 만드는것이다. qstring.h를 포함하여 Qt의 유니코드문자렬들을 사용하게 한다. 편리하게 Elements용의 벡토르용기를 형정의하여 qvaluevector.h머리부파일에 넣는다.

```
typedef QValueVector<Element> ElementVector;
```

Qt는 수많은 용기를 제공하는데 일부는 QValueVector와 같이 값에 기초하고 다른 일부는 지적자에 기초한다(집합클라스참고). 여기서는 하나의 용기를 형정의하고 하나의 ElementVector에 매개 자료모임을 보관한다.

```
const double EPSILON = 0.0000001; // Must be > INVALID.
```

요소들은 정의 값만 가질수 있다. 값을 double로 보관하므로 0과 쉽게 비교할수 있다. 대신에 령에 가까운 값 EPSILON을 지정하고 EPSILON보다 큰 값을 정의 유효값으로 고찰한다.

```
class Element
{
public:
 enum { INVALID = -1 };
 enum { NO_PROPORTION = -1 };
 enum { MAX_PROPOINTS = 3 }; // One proportional point per chart
type
```

Element용으로 3개의 공개렬거형을 정의한다. INVALID는 isValid()함수에서 사용한다. Element들의 고정크기벡토르를 사용하며 그것들에 INVALID값들을 줌으로써 사용하지 않은 Element들을 표식하는데 쓸수 있다. NO_PROPORTION렬거는 사용자가 Element의 표식자를 배치하지 않았다는것을 지정하는데 쓰이며 임의의 정의 비률값은 본문요소의 위치가 캔버스의 크기에 비례하게 취해진다.

매개 표식자의 x, y위치를 보관하였으면 사용자가 기본창문(캔버스)의 크기를 조절할 때마다 본문은 그의 원시(현재 정확하지 않은)위치를 기억하군 한다. 그러므로 절대(x, y)위치를 보관할 대신에 비례위치 즉 x/width와 y/height를 보관한다. 본문을 그리려고 할 때 이 위치들에 현재 폭과 높이를 각각 곱할수 있으며 본문은 크기조절에 관계없이 정확히 위치된다. 실례로 표식자가 x위치 300를 가지고 캔버스가 400화소폭을 가지면 x값의 비례는 300/400 = 0.75이다.

렬거값 MAX_PROPOINTS에는 문제가 있다. 매개 도표형에 대하여 본문표식자의 x와 y비례를 보관하고 고정크기배렬로 이 비례들을 보관하기로 한다. 그러므로 필요한 비례쌍들의 최대수를 지정해야 한다. 이 값은 도표형의 수에 따라 다르며 이것은 Element클라스가 ChartForm클라스에 의해 제공된 도표형의 수와 완전히 련관된다는것을 의미한다.큰 응용프로그람에서는 점들을 보관하는데 벡토르를 사용하며 몇개의 도표형을 사용하는가에 따라서 크기를 동적으로 조절할수 있다.

```
Element( double value = INVALID, QColor valueColor = Qt::gray,
   int valuePattern = Qt::SolidPattern, const QString& label = QString::null,
   QColor labelColor = Qt::black ) {
  init( value, valueColor, valuePattern, label, labelColor );
  for ( int i = 0; i < MAX_PROPOINTS * 2; ++i )
   m_propoints[i] = NO_PROPORTION;
 }
```

구성자는 Element클라스의 모든 성원들에 대한 기정값을 제공한다. 새 요소들은 항상 위치를 가지지 않는다는 표식을 가진다. 구성자처럼 동작하는 set()함수를 제공하므로 init()함수를 사용한다.

```
bool isValid() const { return m_value > EPSILON; }
```

고정크기벡토르에 Element들을 보관하므로 특정한 요소가 유효한가를 검사할수 있어야 한다. 즉 계산에 사용되고 현시되여야 한다. 이것은 isValid()함수에 의해 간단히 달성된다.

```
//element.cpp로부터 발취
double Element::proX( int index ) const
{
   Q_ASSERT(index >= 0 && index < MAX_PROPOINTS);
   return m_propoints[2 * index];
}
```

얻기함수와 설정함수들이 Element의 모든 성원들에 제공된다. proX()와 proY()얻기함수들과 setProX()와 setProY()설정함수들은 비례위치를 적용하는 도표의 형을 식별하는 첨수를 가진다. 이것은 사용자가 같은 자료모임에 대하여 수직막대도표, 수평막대도표 및 원도표에 대하여 따로따로 위치지정된 표식자들을 가질수 있다는것을 의미한다. 또한 Q_ASSERT마크로를 사용하여 도표형첨수에 대한 사전조건검사를 제공한다(4장 4절 참고).

- 자료요소의 읽기와 쓰기

```
// element.h에서 발취
QTextStream &operator<<( QTextStream&, const Element& );
QTextStream &operator>>( QTextStream&, Element& );
```

Element클라스를 더 완성하기 위하여 <<와 >>연산자들을 재정의함으로써 Element들을 본문흐름에 대하여 읽고쓸수 있도록 한다. 간단히 2진스트림을 사용할수 있지만 본문을 사용함으로써 사용자들이 본문편집기에서 자료를 조작하고 스크립트언어에 의하여 자료를 쉽게 생성하고 려과할수 있게 한다.

```
// element.cpp에서 발취
#include "element.h"
#include <qstringlist.h>
#include <qtextstream.h>
```

연산자들의 실현은 qtextstream.h와 qstringlist.h를 포함할것을 요구한다.

```
const char FIELD_SEP = ':';
const char PROPOINT_SEP = ';';
const char XY_SEP = ',';
```

자료를 보관하는데 사용하는 형식은 마당들을 반점으로 구분하고 레코드들을 새행으로 구분한다. 비례점들은 반두점으로 구분하고 x, y쌍은 점으로 구분한다. 마당순서는 값, 값의 색, 값패턴, 표식자의 색, 표식자의 점들, 표식자본문이다. 실례로

```
20:#ff0000:14:#000000:0.767033,0.412946;0,0.75;0,0:Red :with colons:!
70:#00ffff:2:#ffff00:0.450549,0.198661;0.198516,0.125954;0,0.198473:Cyan
35:#0000ff:8:#555500:0.10989,0.299107;0.397032,0.562977;0,0.396947:Blue
55:#ffff00:1:#000080:0.0989011,0.625;0.595547,0.312977;0,0.59542:Yellow
80:#ff00ff:1:#000000:0.518681,0.694196;0.794063,0;0,0.793893:Magenta or Violet
```

Element자료를 읽어들이는 방법으로 인하여 표식자본문에서 나타나는 공백과 마당분리기호들과 관련한 문제는 없다.

```
QTextStream &operator<<( QTextStream &s, const Element &element )
{
 s << element.value() << FIELD_SEP
   << element.valueColor().name() << FIELD_SEP
   << element.valuePattern() << FIELD_SEP
   << element.labelColor().name() << FIELD_SEP;

 for ( int i = 0; i < Element::MAX_PROPOINTS; ++i ) {
  s << element.proX( i ) << XY_SEP << element.proY( i );
  s << ( i == Element::MAX_PROPOINTS - 1 ? FIELD_SEP :
          PROPOINT_SEP );
 }

 s << element.label() << '\n';

 return s;
}
```

요소들을 쓰는 방법은 간단하다. 매개 성원의 뒤에 마당구분기호가 따른다. 점들은 x, y쌍들을 반점(XY_SEP)으로 구분하며 매개 쌍은 PROPOINT_SEP구분기호에 의해 구분된다. 마지막 마당은 표식이고 그 뒤에 새행이 따른다.

```
QTextStream &operator>>( QTextStream &s, Element &element )
{
 QString data = s.readLine();
 element.setValue( Element::INVALID );

 int errors = 0;
 bool ok;

 QStringList fields = QStringList::split( FIELD_SEP, data );
 if ( fields.count() >= 4 ) {
  double value = fields[0].toDouble( &ok );
  if ( !ok )
   errors++;
  QColor valueColor = QColor( fields[1] );
  if ( !valueColor.isValid() )
   errors++;
  int valuePattern = fields[2].toInt( &ok );
  if ( !ok )
   errors++;
  QColor labelColor = QColor( fields[3] );
  if ( !labelColor.isValid() )
```

```
   errors++;
  QStringList propoints = QStringList::split( PROPOINT_SEP, fields[4] );
  QString label = data.section( FIELD_SEP, 5 );

  if ( !errors ) {
   element.set( value, valueColor, valuePattern, label, labelColor );
   int i = 0;
   for ( QStringList::iterator point = propoints.begin();
    i < Element::MAX_PROPOINTS && point != propoints.end();
    ++i, ++point ) {
    errors = 0;
    QStringList xy = QStringList::split( XY_SEP, *point );
    double x = xy[0].toDouble( &ok );
    if ( !ok || x <= 0.0 || x >= 1.0 )
     errors++;
    double y = xy[1].toDouble( &ok );
    if ( !ok || y <= 0.0 || y >= 1.0 )
     errors++;
    if ( errors )
     x = y = Element::NO_PROPORTION;
    element.setProX( i, x );
    element.setProY( i, y );
   }
  }
 }

 return s;
}
```

요소들을 읽어들이기 위하여 한개 레코드(즉 한 행)를 읽는다. QStringList::split()에 의해 자료를 마당들로 분해한다. 표식이 FIELD_SEP 문자들을 포함할수 있으므로 QString::section()에 의해 마지막 마당으로부터 행끝까지의 모든 본문을 꺼낸다. 충분한 마당이 있고 값, 색, 패턴자료가 유효하면 Element::set()에 의하여 이 자료를 요소에 써넣고 그렇지 않으면 요소를 INVALID로 한다. 그다음 점들을 순환한다. x와 y비례는 요소에 대하여 설정된 범위에 있다. 하나 혹은 두 비례가 무효이면 값을 0으로 보관하는데 이것은 무효한 (그리고 범위밖의) 비례점값들을 NO_PROPORTION으로 변경하므로 적합하지 않다.

현재 Element클라스는 요소자료들을 보관하고 조작하고 읽고 쓰는데 충분하지 못하다. 또한 요소집합을 보관하기 위한 요소벡토르형정의도 창조하였다.

이제는 main.cpp와 사용자들이 자료모임을 창조하고 편집하고 볼수 있는 사용자대면부를 만들 준비가 되였다.

3. 주로 간단하게

```
// main.cpp
#include <qapplication.h>
#include "chartform.h"

int main( int argc, char *argv[] )
{
 QApplication app( argc, argv );
```

```
    QString filename;
    if ( app.argc() > 1 ) {
     filename = app.argv()[1];
     if ( !filename.endsWith( ".cht" ) )
      filename = QString::null;
    }

    ChartForm *cf = new ChartForm( filename );
    app.setMainWidget( cf );
    cf->show();

    return app.exec();
   }
```

main()함수를 간단하고 작게 만든다. QApplication객체를 창조하고 거기에 지령행인수들을 넘긴다. 사용자가 도표 mychart.cht를 가지고 프로그람을 호출하게 하여 파일이름이 추가되면 그것을 도표폼구성자를 통하여 넘기도록 한다. 대부분의 작용은 다음에 고찰하는 도표폼에 배치한다.

4. GUI를 현시하기

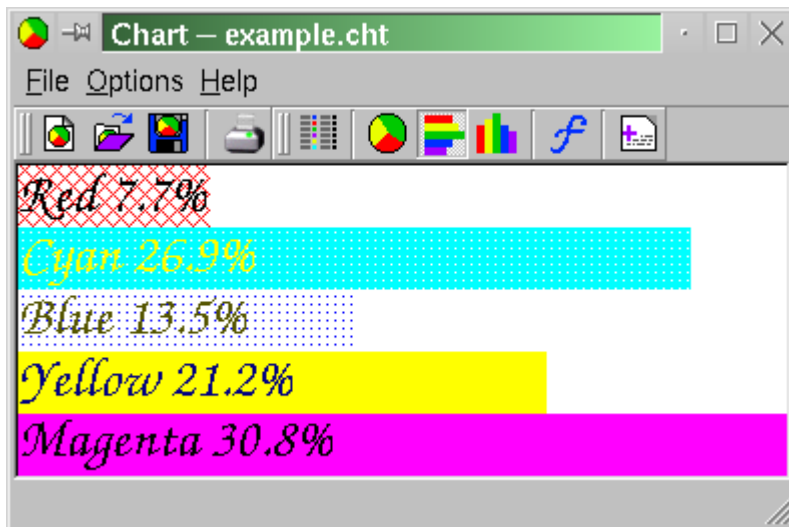



그림 2-17. GUI의 현시

chart응용프로그람은 중심창문부품 즉 편리한 문서중심형식의 CanvasView에 배치된 차림표와 도구띠단추들을 통하여 항목들에 대한 호출을 제공한다.

```
// chartform.h에서 발취
 class ChartForm: public QMainWindow
 {
   Q_OBJECT
  public:
   enum { MAX_ELEMENTS = 100 };
   enum { MAX_RECENTFILES = 9 }; // Must not exceed 9
```

```
 enum ChartType { PIE, VERTICAL_BAR, HORIZONTAL_BAR };
 enum AddValuesType { NO, YES, AS_PERCENTAGE };

 ChartForm( const QString& filename );
 ~ChartForm();

 int chartType() { return m_chartType; }
 void setChanged( bool changed = TRUE ) { m_changed = changed; }
 void drawElements();

 QPopupMenu *optionsMenu; // public로 하는 근거는 canvasview.cpp
                                    를 보시오.

protected:
 virtual void closeEvent( QCloseEvent * );

private slots:
  void fileNew();
  void fileOpen();
  void fileOpenRecent( int index );
  void fileSave();
  void fileSaveAs();
  void fileSaveAsPixmap();
  void filePrint();
  void fileQuit();
  void optionsSetData();
  void updateChartType( QAction *action );
  void optionsSetFont();
  void optionsSetOptions();
  void helpHelp();
  void helpAbout();
  void helpAboutQt();
  void saveOptions();

private:
  void init();
  void load( const QString& filename );
  bool okToClear();
  void drawPieChart( const double scales[], double total,
                     int count );
  void drawVerticalBarChart( const double scales[], double total,
                     int count );
  void drawHorizontalBarChart( const double scales[], double total,
                     int count );

   QString valueLabel( const QString& label, double value,
```

```
                                          double total );
        void updateRecentFiles( const QString& filename );
        void updateRecentFilesMenu();
        void setChartType( ChartType chartType );
        QPopupMenu *fileMenu;
        QAction *optionsPieChartAction;
        QAction *optionsHorizontalBarChartAction;
        QAction *optionsVerticalBarChartAction;
        QString m_filename;
        QStringList m_recentFiles;
        QCanvas *m_canvas;
        CanvasView *m_canvasView;
        bool m_changed;
        ElementVector m_elements;
        QPrinter *m_printer;
        ChartType m_chartType;
        AddValuesType m_addValues;
        int m_decimalPlaces;
        QFont m_font;
    };
```

QMainWindow의 파생클라스 ChartForm를 창조한다. 이 파생클라스는 Q_OBJECT마크로를 사용하여 Qt의 신호-처리부기구를 유지한다.

공개대면부는 아주 작고 현시하고있는 도표형을 얻고 도표는 《변경된것》으로 표시하고(완료할 때 사용자에게 보관하겠는가를 묻는다.) 도표자체를 그리겠는가(drawElements())하는 물음을 얻을수 있다. 또한 선택(option)차림표를 캔버스보기의 상황차림표로 사용하므로 이 차림표를 공개로 만든다.

> QCanvas클라스는 2차원벡토르도형을 그리는데 사용된다. QCanvasView 클라스는 응용프로그람의 GUI에서 캔버스의 보기를 현시하는데 쓰인다. 모든 그리기조작은 캔버스우에서 수행되지만 사건들(실례로 마우스찰칵)은 캔버스보기에서 발생한다.

매개 작용은 비공개처리부에 의해 표시된다. 실례로 fileNew(), optionsSetData() 등이다. 또한 아주 많은 비공개함수와 자료성원들이 있다.

편리성과 콤파일의 속도를 높이기 위하여 도표폼의 실현을 세개 파일 즉 GUI용의 chartform.cpp, 캔버스조종용의 chartform_canvas.cpp 그리고 파일조종용의 chartform_files.cpp로 분리한다.

(1) 도표폼GUI

```
// chartform.cpp에서 발취
 #include "images/file_new.xpm"
 #include "images/file_open.xpm"
 #include "images/options_piechart.xpm"
```

chart에서 사용한 모든 화상들은 images보조등록부에 배치되는 .xpm파일들로 창조된다.

(2) 구성자

```
  ChartForm::ChartForm( const QString& filename )
  : QMainWindow( 0, 0, WDestructiveClose )
 ...
```

```
QAction *fileNewAction;
QAction *fileOpenAction;
QAction *fileSaveAction;
```

매개의 사용자작용에 대하여 QAction지적자를 선언한다. 일부 작용은 구성자의 밖에서 참고되여야 하므로 머리부파일에서 선언된다.

> 대부분의 사용자작용은 차림표항목과 도구띠단추들로서 모두 적합하다. Qt는 차림표와 도구띠에 모두 추가할수 있는 하나의 QAction을 창조한다. 이 수법은 차림표항목과 도구띠단추들을 동기시키고 반복코드를 줄일수 있도록 한다.

```
fileNewAction = new QAction("New Chart", QPixmap( file_new ),
  "&New", CTRL+Key_N, this, "new" );
connect( fileNewAction, SIGNAL( activated() ), this,
SLOT( fileNew() ) );
```

작용을 구성할 때 거기에 이름과 선택적인 그림기호, 차림표본문, 지름건을 준다. 지름건이 요구되지 않으면 0을 준다. 또한 this를 넘기여 그것을 폼의 자식으로 만든다. 사용자가 도구띠단추를 찰칵하거나 차림표항목을 찰칵할 때 activated()신호가 발생된다. 이 신호를 작용의 처리부(우의 코드에서 fileNew())에 connect()한다.

도표형들은 모두 배타적이고 원형도표 혹은 수직막대도표 혹은 수평막대도표를 하나 가질수 있다. 이것은 사용자가 원도표의 차림표항목을 선택하면 원도표의 도구띠단추도 자동적으로 선택되고 다른 도표의 차림표항목들과 도구띠단추들의 선택은 자동적으로 해제되여야 한다는것을 의미한다. 이 동작은 QActionGroup을 창조하고 도표형의 작용들을 그룹안에 배치하여 얻는다.

```
QActionGroup *chartGroup = new QActionGroup( this );
                              // Connected later
chartGroup->setExclusive( TRUE );
```

작용그룹은 폼(this)의 자식으로 되고 배타적인 동작은 setExclusive()호출에 의해 얻는다.

```
optionsPieChartAction = new QAction(
    "Pie Chart", QPixmap( options_piechart ),
    "&Pie Chart", CTRL+Key_I, chartGroup, "pie chart" );
optionsPieChartAction->setToggleAction( TRUE );
```

그룹안의 매개 작용은 다른 작용들과 같은 방법으로 창조된다. 그러나 작용의 부모는 폼이 아니라 그룹이다. 도표형의 작용들이 on/off상태를 가지므로 그 매개에 대하여 setToggleAction(TRUE)를 호출한다. 작용들을 련결하지 않고 그대신에 후에 그룹을 캔버스를 다시그리는 처리부에 련결한다.

> 왜 그룹을 직접 련결하지 않는가? 후에 구성자에서 사용자의 선택들을 읽어들이는데 그 하나가 도표형이다. 그다음 그에 따라 도표형을 설정한다. 그러나 그 시점에서 아직 캔버스를 창조하지 않거나 자료가 없으므로 우리가 할 일은 캔버스형의 도구띠단추들을 절환하는것이지만 그때 존재하지 않는 캔버스를 실제로 그리지 않고 캔버스형을 설정한 후에 그룹을 련결한다.

사용자의 작용을 모두 창조한 다음 도구띠와 차림표선택들을 창조하여 사용자가 그것들을 호출할수 있게 한다.

```
QToolBar* fileTools = new QToolBar( this, "file operations" );
fileTools->setLabel( "File Operations" );
```

```
    fileNewAction->addTo( fileTools );
    fileOpenAction->addTo( fileTools );
    fileSaveAction->addTo( fileTools );
  ...
    fileMenu = new QPopupMenu( this );
    menuBar()->insertItem( "&File", fileMenu );
    fileNewAction->addTo( fileMenu );
    fileOpenAction->addTo( fileMenu );
    fileSaveAction->addTo( fileMenu );
```

도구띠의 작용과 차림표선택들은 QAction들로부터 간단히 창조된다.

사용자들이 편리하게 마지막 창문위치와 크기를 되살리고 최근에 사용한 파일들을 렬거한다. 이것은 응용프로그람이 닫길 때 설정값들을 써넣고 폼을 구성할 때 그것들을 읽어들임으로써 달성된다.

```
    QSettings settings;
    settings.insertSearchPath( QSettings::Windows, WINDOWS_REGISTRY );
    int windowWidth = settings.readNumEntry( APP_KEY + "WindowWidth", 460 );
    int windowHeight = settings.readNumEntry( APP_KEY + "WindowHeight", 530 );
    int windowX = settings.readNumEntry( APP_KEY + "WindowX", -1 );
    int windowY = settings.readNumEntry( APP_KEY + "WindowY", -1 );
    setChartType(ChartType(settings.readNumEntry(APP_KEY + "ChartType", int(PIE))));
    m_font = QFont( "Helvetica", 18, QFont::Bold );
    m_font.fromString( settings.readEntry( APP_KEY + "Font", m_font.toString() ) );
    for ( int i = 0; i < MAX_RECENTFILES; ++i ) {
     QString filename = settings.readEntry( APP_KEY + "File" + QString::number( i + 1 ) );
     if ( !filename.isEmpty() )
      m_recentFiles.push_back( filename );
    }
    if ( m_recentFiles.count() )
     updateRecentFilesMenu();
```

QSettings클라스는 가동환경에 의존하지 않는 방법으로 사용자의 설정을 처리한다. 간단히 설정값들을 읽고쓰고 QSettings이 가동환경의존관계를 조종한다. insertSearchPath()호출은 아무 일도 하지 않는다.

readNumEntry()호출을 사용하여 도표폼의 마지막 크기와 위치를 얻으며 이 함수가 처음으로 실행되면 기정값들을 제공한다. 도표형은 옹근수로 얻어지고 ChartType렬거형으로 강제변환된다. 기정표식서체를 창조하고 Font설정을 읽어들이며 필요하다면 기정값에 기초하여 창조한다.

QSettings가 문자렬목록을 취급할수 있더라도 최근에 사용한 매개 파일을 개별적인 항목에 보관하도록 선택하여 환경설정을 간단히 손으로 편집하게 한다. 매개의 가능한 파일항목("File1"로부터 "File9"까지)을 읽어들이고 비지 않은 매개 항목을 최근에 사용한 파일들의 목록에 추가한다. 최근에 사용한 파일들이 하나이상 있으면 updateRecentFilesMenu()를 호출하여 File차림표를 갱신한다.

```
    connect( chartGroup, SIGNAL( selected(QAction*) ),
      this, SLOT( updateChartType(QAction*) ) );
```

도표형을 설정하였으므로 도표그룹을 updateChartType()처리부에 련결하는것이 안전하다.

```
resize( windowWidth, windowHeight );
if ( windowX != -1 || windowY != -1 )
 move( windowX, windowY );
```

그리고 창문의 크기와 위치를 알고있으므로 도표폼의 창문크기를 조절하고 이동할 수 있다.

```
m_canvas = new QCanvas( this );
m_canvas->resize( width(), height() );
m_canvasView = new CanvasView( m_canvas, &m_elements, this );
setCentralWidget( m_canvasView );
m_canvasView->show();
```

새로운 QCanvas를 창조하고 그 크기를 도표폼창문의 의뢰령역크기로 설정한다. 또한 CanvasView (QCanvasView의 파생클라스)를 창조하고 QCanvas를 현시한다. 캔버스보기를 도표폼의 기본창문부품으로 만들고 그것을 표시한다.

```
if ( !filename.isEmpty() )
   load( filename );
else {
 init();
 m_elements[0].set( 20, red, 14, "Red" );
 m_elements[1].set( 70, cyan, 2, "Cyan",   darkGreen );
 m_elements[2].set( 35, blue,   11, "Blue" );
 m_elements[3].set( 55, yellow,  1, "Yellow", darkBlue );
 m_elements[4].set( 80, magenta, 1, "Magenta" );
 drawElements();
}
```

적재할 파일이 있으면 적재하고 없으면 요소벡토르들을 초기화하고 견본도표를 그린다.

```
statusBar()->message( "Ready", 2000 );
```

구성자에서 statusBar()를 호출하는것이 아주 중요하다. 이 호출은 상태띠를 기본창문용으로 창조한다.

① init()

```
void ChartForm::init()
{
   setCaption( "Chart" );
   m_filename = QString::null;
   m_changed = FALSE;

   m_elements[0]  = Element( Element::INVALID, red );
   m_elements[1]  = Element( Element::INVALID, cyan );
   m_elements[2]  = Element( Element::INVALID, blue );
...
```

폼이 구성될 때 그리고 사용자가 현존자료모임을 적재하거나 새로운 자료모임을 창조할 때마다 init()를 사용하여 캔버스와 요소들(ElementVector에서 m_elements)을 초기화한다.

제목을 재설정하고 현재파일이름을 QString::null로 설정한다. 또한 요소벡토르를 무효요소들로 채운다. 이것은 필요하지 않지만 매개 요소에 다른 색을 주는것은 사용자가 값들을 입

력할 때 매개가 유일한 색(변경가능)을 이미 가지고있으므로 사용자에게 더 편리하다.

(3) 파일조종작용

① okToClear()

```
bool ChartForm::okToClear()
{
 if ( m_changed ) {
  QString msg;
  if ( m_filename.isEmpty() )
   msg = "Unnamed chart ";
  else
   msg = QString( "Chart '%1'\n" ).arg( m_filename );
  msg += "has been changed.";

  int x = QMessageBox::information( this, "Chart -- Unsaved Changes",
   msg, "&Save", "Cancel", "&Abandon",
   0, 1 );
  switch( x ) {
   case 0: // Save
    fileSave();
    break;
   case 1: // Cancel
   default:
    return FALSE;
   case 2: // Abandon
    break;
  }
 }
 return TRUE;
}
```

okToClear()함수는 보관하지 않은 자료가 있을 때 그것들을 보관하겠는가를 사용자에게 묻는데 사용된다. 그것을 여러개의 다른 함수들이 사용한다.

② fileNew()

```
void ChartForm::fileNew()
{
 if ( okToClear() ) {
  init();
  drawElements();
 }
}
```

사용자가 fileNew()작용을 호출할 때 okToClear()를 호출하여 보관하지 않은 자료를 보관할 기회를 준다. 보관하거나 무시하거나 보관할 자료가 없으면 요소벡토르를 다시 초기화하고 기정도표를 그린다.

또한 optionsSetData()를 호출하여 사용자가 값, 색 등을 창조하고 편집할수 있는 대화칸을 펼칠수 있어야 한다. 응용프로그람을 실행하고 optionsSetData()에로의 호출을 추가하고 자기가 좋아하는것을 보려고 할수 있다.

③ fileOpen()

```
void ChartForm::fileOpen()
{
 if ( !okToClear() )
  return;

 QString filename = QFileDialog::getOpenFileName(QString::null,
                 "Charts (*.cht)", this, "file open", "Chart -- File
                 Open" );
 if ( !filename.isEmpty() )
  load( filename );
 else
  statusBar()->message( "File Open abandoned", 2000 );
}
```

그것이 okToClear()인가 검사하고 그러면 정적함수 QFileDialog::getOpenFileName()에 의해 자용자가 적재하려는 파일의 이름을 얻고 load()를 호출한다.

④ fileSaveAs()

```
void ChartForm::fileSaveAs()
{
 QString filename = QFileDialog::getSaveFileName(QString::null,
      "Charts (*.cht)", this, "file save as", "Chart -- File Save As" );
 if ( !filename.isEmpty() ) {
  int answer = 0;
  if ( QFile::exists( filename ) )
   answer = QMessageBox::warning(this, "Chart -- Overwrite File",
       QString( "Overwrite\n\'%1\'?" ).arg( filename ),
       "&Yes", "&No", QString::null, 1, 1 );
  if ( answer == 0 ) {
   m_filename = filename;
   updateRecentFiles( filename );
   fileSave();
   return;
  }
 }
 statusBar()->message( "Saving abandoned", 2000 );
}
```

이 함수는 정적함수 QFileDialog::getSaveFileName()를 호출하여 자료를 보관하려는 파일의 이름을 얻는다. 파일이 존재하면 QMessageBox::warning()에 의하여 사용자에게 통지하고 보관을 무시할수 있는 기회를 준다. 파일이 보관하려고 한다면 최근에 연 파일목록을 갱신하고 fileSave()(파일조종에서 취급)를 호출하여 보관한다.

(4) 최근에 연 파일목록의 관리

```
QStringList m_recentFiles;
```

최근에 연 파일들의 목록을 문자렬목록을 보관한다.

```
void ChartForm::updateRecentFilesMenu()
{
```

```
for ( int i = 0; i < MAX_RECENTFILES; ++i ) {
 if ( fileMenu->findItem( i ) )
  fileMenu->removeItem( i );
 if ( i < int(m_recentFiles.count()) )
  fileMenu->insertItem( QString( "&%1 %2" ). arg( i + 1 ).arg( m_recentFiles[i] ),
     this, SLOT( fileOpenRecent(int) ), 0, i );
}
}
```

이 함수는 사용자가 현존파일을 열거나 새로운 파일을 보관할 때마다 보통 updateRecentFiles()를 통하여 호출된다. 문자렬목록안의 매개 파일에 대하여 새로운 차림표항목을 삽입한다. 매개의 파일이름앞에 밑줄있는 번호(1 ～9)를 배치하여 건반호출기능을 추가한다.(실례로 Alt+F, 2는 목록의 둘째 파일을 연다.) 차림표항목에 문자렬목록에서 항목의 첨수위치와 같은 id를 주고 매개 차림표항목을 fileOpenRecent()처리부에 련결한다. 낡은 파일차림표항목들은 매개의 가능한 최근파일차림표항목id를 통하여 동시에 탐지된다. 이것은 다른 파일차림표항목들이 Qt에 의해 창조된 id를 가지고있으므로 제대로 동작한다. (그 모두는 0보다 작다.) 그런가 하면 우리가 창조한 차림표항목들은 모두 0이상의 id를 가진다.

```
void ChartForm::updateRecentFiles( const QString& filename )
{
 if ( m_recentFiles.find( filename ) != m_recentFiles.end() )
  return;

 m_recentFiles.push_back( filename );
 if ( m_recentFiles.count() > MAX_RECENTFILES )
  m_recentFiles.pop_front();

 updateRecentFilesMenu();
}
```

이것은 사용자가 현존파일들을 열거나 새로운 파일들을 보관할 때 호출된다. 파일이 이미 목록에 있으면 단순히 되돌아간다. 그렇지 않으면 파일이 목록의 끝에 추가되고 목록이 너무 크면(파일이 10개이상이면) 첫째의 제일 낡은 파일이 삭제된다. updateRecentFilesMenu()이 그다음 호출되여 File차림표에 최근에 사용한 파일들의 목록을 다시 창조한다.

```
void ChartForm::fileOpenRecent( int index )
{
 if ( !okToClear() )
  return;

 load( m_recentFiles[index] );
}
```

사용자가 최근에 연 파일을 선택할 때 선택한 파일의 차림표id에 의해 fileOpenRecent()처리부가 호출된다. 파일차림표 id들을 m_recentFiles목록에서 파일들의 첨수위치들과 같이 만들었으므로 차림표항목 id에 의해 색인화된 파일을 간단히 적재할수 있다.

(5) 완료

```
void ChartForm::fileQuit()
{
 if ( okToClear() ) {
  saveOptions();
  qApp->exit( 0 );
 }
}
```

사용자가 완료할 때 보관하지 않은 자료(실례로 창문의 크기와 위치, 도표형 등)를 보관할 기회를 완료하기전에 준다(okToClear()).

```
void ChartForm::saveOptions()
{
 QSettings settings;
 settings.insertSearchPath( QSettings::Windows, WINDOWS_REGISTRY );
 settings.writeEntry( APP_KEY + "WindowWidth", width() );
 settings.writeEntry( APP_KEY + "WindowHeight", height() );
 settings.writeEntry( APP_KEY + "WindowX", x() );
 settings.writeEntry( APP_KEY + "WindowY", y() );
 settings.writeEntry( APP_KEY + "ChartType", int(m_chartType) );
 settings.writeEntry( APP_KEY + "AddValues", int(m_addValues) );
 settings.writeEntry( APP_KEY + "Decimals", m_decimalPlaces );
 settings.writeEntry( APP_KEY + "Font", m_font.toString() );
 for ( int i = 0; i < int(m_recentFiles.count()); ++i )
  settings.writeEntry( APP_KEY + "File" + QString::number( i + 1 ),
       m_recentFiles[i] );
}
```

QSettings를 리용하여 사용자의 선택값들을 보관하는것은 간단하다.

(6) 사용자정의대화칸

사용자가 일부 선택을 수동적으로 설정하고 값들과 값들의 색 등을 창조하고 편집할수 있게 하려고 한다.

```
void ChartForm::optionsSetOptions()
{
 OptionsForm *optionsForm = new OptionsForm( this );
 optionsForm->chartTypeComboBox->setCurrentItem( m_chartType );
 optionsForm->setFont( m_font );
 if ( optionsForm->exec() ) {
  setChartType( ChartType(
    optionsForm->chartTypeComboBox->currentItem()) );
  m_font = optionsForm->font();
  drawElements();
 }
 delete optionsForm;
}
```

선택값들을 설정하기 위한 폼은 사용자정의OptionsForm에 의해 제공된다. 선택폼은 실례를 창조하고 그의 모든 GUI요소들을 관련한 설정값들로 설정하고 사용자가 OK를 찰칵하면 (exec()가 true값을 돌려준다.) GUI요소들로부터 설정값들을 다시 읽어들인다.

```
void ChartForm::optionsSetData()
{
 SetDataForm *setDataForm = new SetDataForm( &m_elements,
m_decimalPlaces, this );
 if ( setDataForm->exec() ) {
  m_changed = TRUE;
  drawElements();
 }
 delete setDataForm;
}
```

도표자료를 창조하고 편집하기 위한 폼은 사용자정의SetDataForm에 의해 제공된다. 작업하려고 하는 자료구조를 넘기고 대화칸은 자료구조자체의 제시를 조종한다. 사용자가 OK를 찰칵하면 대화칸이 자료구조를 갱신하고 exec()는 true값을 돌려준다. 사용자가 자료를 변경하였을 때 optionsSetData()에서 해야 할 일은 도표를 변경된것으로 표식하고 drawElements()를 호출하여 새로운 갱신값으로 도표를 다시 그리는것이다.

5. 캔버스조종

원도표의 부채형토막 (혹은 막대도표의 막대들), 그리고 표식자들을 캔버스우에 그린다. 캔버스는 캔버스보기를 통하여 사용자에게 표시된다. drawElements()함수는 필요할 때 캔버스를 다시 그리기 위하여 호출된다. 다음의 코드는 chartform_canvas.cpp에서 발취한것이다.

(1) drawElements()

```
void ChartForm::drawElements()
{
 QCanvasItemList list = m_canvas->allItems();
 for ( QCanvasItemList::iterator it = list.begin(); it != list.end(); ++it )
  delete *it;
```

그리려는 도표형에 의존하는 비례결수를 계산한다.

```
 double biggest = 0.0;
 int count = 0;
 double total = 0.0;
 static double scales[MAX_ELEMENTS];

 for ( int i = 0; i < MAX_ELEMENTS; ++i ) {
  if ( m_elements[i].isValid() ) {
   double value = m_elements[i].value();
   count++;
   total += value;
   if ( value > biggest )
    biggest = value;
   scales[i] = m_elements[i].value() * scaleFactor;
  }
 }

 if ( count ) {
   // 2nd loop because of total and biggest
```

```
for ( int i = 0; i < MAX_ELEMENTS; ++i )
 if ( m_elements[i].isValid() )
  if ( m_chartType == PIE )
   scales[i] = (m_elements[i].value() * scaleFactor) / total;
  else
   scales[i] = (m_elements[i].value() * scaleFactor) / biggest;
```

값의 개수, 최대값과 총계를 알아야 정확히 비례되는 부채형이나 막대들을 창조할수 있다. scales배렬에 비례값들을 보관한다.

```
switch ( m_chartType ) {
 case PIE:
  drawPieChart( scales, total, count );
  break;
 case VERTICAL_BAR:
  drawVerticalBarChart( scales, total, count );
  break;
 case HORIZONTAL_BAR:
  drawHorizontalBarChart( scales, total, count );
  break;
}
}
```

캔버스를 update()하여 변경이 나타나게 한다.

① drawHorizontalBarChart()

비례값들의 배렬, 총계값(필요할 때 퍼센트를 계산하고 그리기 위하여), 값들의 개수가 필요하다.

```
double width = m_canvas->width();
double height = m_canvas->height();
int proheight = int(height / count);
int y = 0;
```

캔버스의 폭과 높이를 얻고 비례높이(proheight)를 계산하고 초기의 y위치를 0으로 설정한다.

```
QPen pen;
pen.setStyle( NoPen );
```

매개 막대(직4각형)를 그리는데 사용할 펜을 창조하고 그것을 NoPen으로 설정하여 테두리선이 그려지지 않게 한다.

```
for ( int i = 0; i < MAX_ELEMENTS; ++i ) {
 if ( m_elements[i].isValid() ) {
  int extent = int(scales[i]);
```

요소벡토르에서 무효요소들을 뛰여넘으면서 매개 요소를 순환한다. 매개 막대의 범위(그 길이)는 단순히 그 비례값(scaled value)이다.

```
  QCanvasRectangle *rect = new QCanvasRectangle(0, y, extent,
proheight, m_canvas );
  rect->setBrush( QBrush( m_elements[i].valueColor(),
     BrushStyle(m_elements[i].valuePattern()) ) );
  rect->setPen( pen );
  rect->setZ( 0 );
  rect->show();
```

매개 막대에 대하여 x위치가 0(이것은 매개 막대가 왼쪽에서 시작하는 수평막대도표이므로), y값이 0으로 시작하고 매개 막대를 그릴 때 그 높이만큼 커지는 새로운 QCanvasRectangle을 창조한다. 그려지는 높이는 막대의 높이와 막대를 그리는 캔버스의 높이에 의해 결정된다. 그다음 사용자가 요소에 대하여 지정한 색과 패턴으로 막대의 솔을 설정하고 펜을 처음에 창조한 펜(즉 NoPen)으로 설정한다. 그리고 Z차원에서 위치 0에 막대를 배치한다. 끝으로 show()를 호출하여 캔버스우에 막대를 그린다.

```
        QString label = m_elements[i].label();
        if ( !label.isEmpty() || m_addValues != NO ) {
         double proX = m_elements[i].proX( HORIZONTAL_BAR );
         double proY = m_elements[i].proY( HORIZONTAL_BAR );
         if ( proX < 0 || proY < 0 ) {
          proX = 0;
          proY = y / height;
         }
```

사용자가 그 요소에 대한 표식을 지정하거나 값(혹은 퍼센트)들을 표시하도록 요구하면 캔버스본문항목도 그린다. 매개 캔버스본문항목안의 대응하는 요소첨수를 보관하여야 하므로 자체의 CanvasText클라스를 창조한다. 요소로부터 x와 y의 비례값들을 꺼낸다. 0보다 작으면 사용자가 위치를 지정하지 않았으므로 그 위치를 계산한다. 표식자의 x값을 0(왼쪽), y값을 막대의 웃끝으로 설정한다. (그리하여 표식자의 왼쪽웃구석이 바로 x, y위치로 된다.)

```
         label = valueLabel( label, m_elements[i].value(), total );
```

그다음 표식자본문을 포함하는 문자렬을 돌려주는 valueLabel()을 호출한다. (valueLabel()함수는 사용자가 적당한 선택들을 설정하였다면 본문표식자에 값이나 퍼센트를 추가한다.)

```
         CanvasText *text = new CanvasText( i, label, m_font, m_canvas );
         text->setColor( m_elements[i].labelColor() );
         text->setX( proX * width );
         text->setY( proY * height );
         text->setZ( 1 );
         text->show();
         m_elements[i].setProX( HORIZONTAL_BAR, proX );
         m_elements[i].setProY( HORIZONTAL_BAR, proY );
```

그다음 요소벡토르에서 이 요소의 첨수, 표식자, 사용하려는 서체와 캔버스를 넘기여 항목을 창조한다. 본문항목의 본문색을 사용자가 지정한 색으로 설정하고 항목의 x와 y위치들을 캔버스의 폭과 높이에 비례되게 설정한다. Z차원을 1로 설정하여 본문항목이 늘 막대(그 Z차원은 0)우에(정면에) 있게 한다. show()를 호출하여 캔버스에 본문항목을 그리고 요소의 상대적인 x와 y위치들을 설정한다.

```
        }
        y += proheight;
```

막대와 그 표식자를 그린 후에 y를 비례높이만큼 증가하고 다음 요소를 그릴 준비를 한다.

```
      }
     }
    }
```

(2) QCanvasText의 파생클라스 만들기

```
   // canvastext.h.로부터 발취
  class CanvasText : public QCanvasText
  {
```

```
public:
 enum { CANVAS_TEXT = 1100 };

 CanvasText( int index, QCanvas *canvas )
  : QCanvasText( canvas ), m_index( index ) {}
 CanvasText( int index, const QString& text, QCanvas *canvas )
  : QCanvasText( text, canvas ), m_index( index ) {}
 CanvasText( int index, const QString& text, QFont font, QCanvas *canvas )
  : QCanvasText( text, font, canvas ), m_index( index ) {}

 int index() const { return m_index; }
 void setIndex( int index ) { m_index = index; }

 int rtti() const { return CANVAS_TEXT; }

private:
 int m_index;
};
```

CanvasText파생클라스는 QCanvasText의 아주 단순한 특수화이다. 우리가 할 일은 이 본문항목과 련관된 요소벡토르첨수를 보관하는 하나의 비공개성원 m_index를 추가하고 이 값을 위한 얻기함수와 설정함수들을 제공하는것이다.

(3) QCanvasView의 파생클라스화

```
// canvasview.h로부터 발취
class CanvasView : public QCanvasView
{
 Q_OBJECT
public:
 CanvasView( QCanvas *canvas, ElementVector *elements,
   QWidget* parent = 0, const char* name = "canvas view", WFlags f = 0 )
      : QCanvasView( canvas, parent, name, f ), m_movingItem(0),
        m_elements( elements ) {}

protected:
 void viewportResizeEvent( QResizeEvent *e );
 void contentsMousePressEvent( QMouseEvent *e );
 void contentsMouseMoveEvent( QMouseEvent *e );
 void contentsContextMenuEvent( QContextMenuEvent *e );

private:
 QCanvasItem *m_movingItem;
 QPoint m_pos;
 ElementVector *m_elements;
};
```

다음과 같은 기능을 처리할수 있도록의 파생클라스를 만들 필요가 있다.

① 상황차림표요구

② 폼크기조절

③ 임의의 위치로 표식자를 끌고가기

이것들을 유지하기 위하여 이동하고있는 지시자와 그 마지막위치를 보관하며 요소벡토르의 지적자도 보관한다.

- 상황차림표의 유지

```
 // canvasview.cpp로부터 발취
void CanvasView::contentsContextMenuEvent( QContextMenuEvent * )
{
 ((ChartForm*)parent())->optionsMenu->exec( QCursor::pos() );
}
```

사용자가 상황차림표를 호출할 때(실례로 대부분의 가동환경에서 오른단추를 찰칵하여) 캔버스보기의 부모(도표폼)를 정확한 형으로 강제변환하고 유표의 위치에서 선택차림표를 실행한다.

- 크기조종

```
void CanvasView::viewportResizeEvent( QResizeEvent *e )
{
 canvas()->resize( e->size().width(), e->size().height() );
 ((ChartForm*)parent())->drawElements();
}
```

크기를 조절하기 위하여 단순히 캔버스보기가 폼의 의뢰령역의 폭과 높이를 표시하는 캔버스의 크기를 조절하고 drawElements()를 호출하여 도표를 다시 그린다. drawElements()가 캔버스의 폭과 높이에 상응하게 그리므로 도표는 정확히 그려진다.

- 주어진 위치로 끌기

사용자가 표식자를 일정한 위치로 끌고가려고 할 때 그것을 찰칵하고 새 위치에로 끌고가서 놓는다.

```
void CanvasView::contentsMousePressEvent( QMouseEvent *e )
{
 QCanvasItemList list = canvas()->collisions( e->pos() );
 for ( QCanvasItemList::iterator it = list.begin(); it != list.end(); ++it )
  if ( (*it)->rtti() == CanvasText::CANVAS_TEXT ) {
   m_movingItem = *it;
   m_pos = e->pos();
   return;
  }
 m_movingItem = 0;
}
```

사용자가 마우스를 찰칵할 때 유표위치에 캔버스항목(있다면)들의 목록을 창조한다. 그다음 목록을 순환하여 CanvasText항목을 찾으면 그것을 이동할 항목으로 설정하고 그 위치를 기록한다. 그렇지 않으면 이동할 항목이 없는것으로 설정한다.

```
void CanvasView::contentsMouseMoveEvent( QMouseEvent *e )
{
 if ( m_movingItem ) {
  QPoint offset = e->pos() - m_pos;
  m_movingItem->moveBy( offset.x(), offset.y() );
  m_pos = e->pos();
  ChartForm *form = (ChartForm*)parent();
```

```
    form->setChanged( TRUE );
    int chartType = form->chartType();
    CanvasText *item = (CanvasText*)m_movingItem;
    int i = item->index();

    (*m_elements)[i].setProX( chartType, item->x() / canvas()->width() );
    (*m_elements)[i].setProY( chartType, item->y() / canvas()->height() );

    canvas()->update();
  }
}
```

사용자가 마우스를 끌기할 때 이동사건이 발생된다. 이동할 항목이 있으면 마지막 마우스위치로부터의 편차를 계산하고 항목을 그 편차량만큼 이동한다. 새 위치를 마지막 위치로서 기록한다. 도표가 현재 변경되였으므로 setChanged()를 호출하여 사용자가 완료하거나 현존도표를 적재하거나 새 도표를 창조하려고 하는 경우에 보관하겠는가 하는 재촉문이 표시되게 한다. 또한 현재 도표형에 대하여 요소의 x와 y의 비례위치들을 폭과 높이에 각각 비례하는 현재의 x와 y 위치들로 갱신한다. 매개 캔버스항목을 창조할 때 거기에 대응하는 요소의 첨수위치를 넘겨야 하므로 갱신하여야 할 요소를 알고있다. QCanvasText의 파생클라스를 만들어서 이 첨수값을 설정하고 얻을수 있다. 끝으로 update()를 호출하여 캔버스를 다시 그린다.

QCanvas는 시각적인 표시가 없다. 캔버스의 내용을 알려면 QCanvasView를 창조하여 캔버스를 표시하여야 한다. 매개 항목에 대하여 show()를 호출하고 QCanvas::update()를 호출하여야만 그것들이 캔버스보기에 나타난다. 기정으로 QCanvas의 배경색은 백색이고 기정으로 캔버스우에 그려진 도형들 실례로 QCanvasRectangle, QCanvasEllipse 등은 백색으로 색칠되므로 백색이 아닌 솔을 설정하는것이 좋다.

6. 파일조종

다음의 코드는 chartform_files.cpp에서 발취한것이다.

① 도표자료의 읽어들이기

```
void ChartForm::load( const QString& filename )
{
  QFile file( filename );
  if ( !file.open( IO_ReadOnly ) ) {
    statusBar()->message( QString( "Failed to load \'%1\'" ).arg( filename ), 2000 );
    return;
  }

  init(); // 색들을 가지고있는가 확인
  m_filename = filename;
  QTextStream ts( &file );
  Element element;
  int errors = 0;
  int i = 0;
  while ( !ts.eof() ) {
```

```
  ts >> element;
  if ( element.isValid() )
   m_elements[i++] = element;
 file.close();
 setCaption( QString( "Chart -- %1" ).arg( filename ) );
 updateRecentFiles( filename );

 drawElements();
 m_changed = FALSE;
}
```

자료모임의 적재는 아주 간단하다. 파일을 열고 본문스트림을 창조한다. 읽어들일 자료가 있으면 요소를 element에 흘러보내고 그것이 유효이면 m_elements벡토르에 삽입한다. 모든 세부는 Element 클라스에 의해 조종된다. 그다음 파일을 닫고 제목과 최근파일 목록을 갱신한다. 끝으로 도표를 그리고 그것을 변경하지 않은것으로 표식한다.

② 도표자료의 써넣기

```
void ChartForm::fileSave()
{
 QFile file( m_filename );
 if ( !file.open( IO_WriteOnly ) ) {
  statusBar()->message( QString( "Failed to save \'%1\'" ).arg( m_filename ), 2000 );
  return;
 }
 QTextStream ts( &file );
 for ( int i = 0; i < MAX_ELEMENTS; ++i )
  if ( m_elements[i].isValid() )
   ts << m_elements[i];

 file.close();

 setCaption( QString( "Chart -- %1" ).arg( m_filename ) );
 statusBar()->message( QString( "Saved \'%1\'" ).arg( m_filename ), 2000 );
 m_changed = FALSE;
}
```

자료보관도 간단하다. 파일을 열고 본문스트림을 창조한 다음 매개 유효요소들을 본문스트림에 보낸다. 모든 세부는 Element클라스에 의해 조종된다.

7. 자료취급

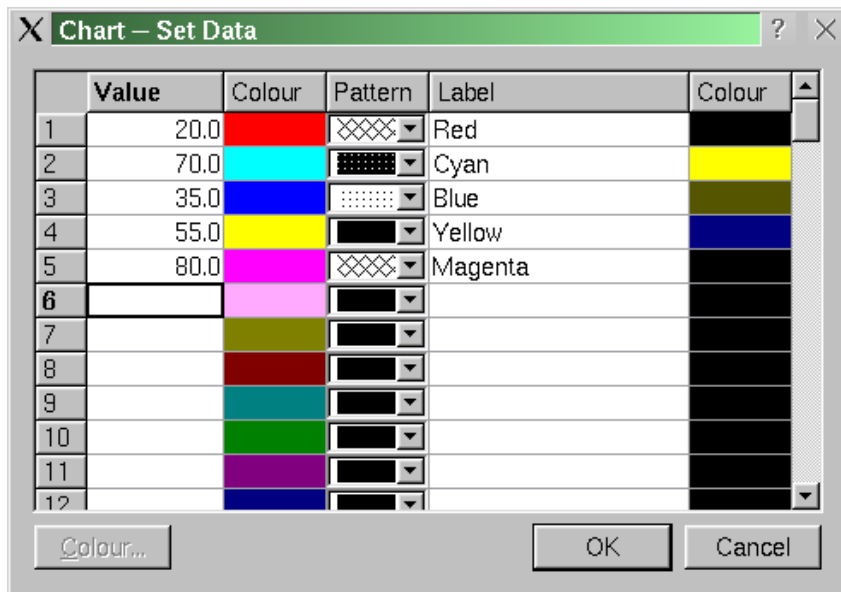



그림 2-18. 자료의 취급

자료설정대화칸은 사용자가 값들을 추가 및 편집하게 하며 값들을 현시하는데 사용할 색과 패턴을 선택하게 한다. 또한 사용자들은 표식자본문을 입력하고 매개 표식자의 색을 선택할수 있다.

```
// setdataform.h로부터 발취
class SetDataForm: public QDialog
{
 Q_OBJECT
public:
 SetDataForm( ElementVector *elements, int decimalPlaces,  QWidget *parent = 0,const char *name = "set data form", bool modal = TRUE, WFlags f = 0 );
 ~SetDataForm() {}
public slots:
 void setColor();
 void setColor( int row, int col );
 void currentChanged( int row, int col );
 void valueChanged( int row, int col );
protected slots:
 void accept();
private:
 QTable *table;
 QPushButton *colorPushButton;
 QPushButton *okPushButton;
 QPushButton *cancelPushButton;
protected:
 QVBoxLayout *tableButtonBox;
 QHBoxLayout *buttonBox;
```

```
private:
 ElementVector *m_elements;
 int m_decimalPlaces;
};
```

머리부파일은 간단하다. 구성자는 요소벡토르의 지적자를 가지고있으므로 이것을 사용하여 대화칸을 현시하고 자료를 직접 편집할수 있다.

```
// setdataform.cpp로부터 발취
#include "images/pattern01.xpm"
#include "images/pattern02.xpm"
```

자그마한 .XPM화상을 참조하여 Qt가 제공하는 매개 솔패턴을 보여준다. 패턴복합칸에서 이것들을 사용한다.

(1) 구성자

```
SetDataForm::SetDataForm( ElementVector *elements, int decimalPlaces,
   QWidget* parent,  const char* name, bool modal, WFlags f )
   : QDialog( parent, name, modal, f )
{
 m_elements = elements;
 m_decimalPlaces = decimalPlaces;
```

QDialog기초클라스에 대부분의 인수를 넘긴다. 요소벡토르지적자와 현시하려는 10진자리수들을 성원변수들에 할당하여 그것들을 SetDataForm의 모든 성원함수들에서 호출할수 있도록 한다.

```
 setCaption( "Chart -- Set Data" );
 resize( 540, 440 );
```

대화칸의 제목을 설정하고 크기를 조절한다.

```
 tableButtonBox = new QVBoxLayout( this, 11, 6, "table button box layout" );
```

폼의 배치는 아주 간단하다. 단추들은 수평배치관리자에 모두 그룹화되고 표와 단추의 배치는 tableButtonBox배치관리자에 의하여 수직으로 그룹화된다.

```
 table = new QTable( this, "data table" );
 table->setNumCols( 5 );
 table->setNumRows( ChartForm::MAX_ELEMENTS );
 table->setColumnReadOnly( 1, TRUE );
 table->setColumnReadOnly( 2, TRUE );
 table->setColumnReadOnly( 4, TRUE );
 table->setColumnWidth( 0, 80 );
 table->setColumnWidth( 1, 60 ); // 1렬과 4렬은 같아야 한다
 table->setColumnWidth( 2, 60 );
 table->setColumnWidth( 3, 200 );
 table->setColumnWidth( 4, 60 );
 QHeader *th = table->horizontalHeader();
 th->setLabel( 0, "Value" );
 th->setLabel( 1, "Color" );
 th->setLabel( 2, "Pattern" );
 th->setLabel( 3, "Label" );
 th->setLabel( 4, "Color" );
 tableButtonBox->addWidget( table );
```

5개의 렬과 요소벡토르안의 요소수와 같은 행수를 가지는 QTable을 새로 창조한다. 색과 패턴렬들을 읽기전용으로 만든다. 이것은 사용자의 입력을 막기 위해서이다. 사용자가 색을 찰칵하거나 Color단추를 찰칵하여 색을 변경할수 있게 만든다. 패턴은 복합칸안에 있으며 사용자가 다른 패턴을 선택하여 간단히 변경할수 있다. 다음으로 초기폭을 적당히 설정하고 매개 렬의 표식자를 삽입하고 끝으로 표식자를 tableButtonBox배치관리자에 추가한다.

```
buttonBox = new QHBoxLayout( 0, 0, 6, "button box layout" );
```

단추들을 보관하기 위한 수평칸배치관리자를 창조한다.

```
colorPushButton = new QPushButton( this, "color button" );
colorPushButton->setText( "&Color..." );
colorPushButton->setEnabled( FALSE );
buttonBox->addWidget( colorPushButton );
```

색단추를 창조하고 거기에 buttonBox배치관리자를 추가한다. 단추를 금지하고 색세포우에 실제로 초점이 있을 때만 그것을 허용한다.

```
QSpacerItem *spacer = new QSpacerItem( 0, 0,
QSizePolicy::Expanding,QSizePolicy::Minimum );
buttonBox->addItem( spacer );
```

색단추를 OK와 Cancel단추로부터 분리하기 위하여 다음에 수축자를 창조하고 그것을 buttonBox배치관리자에 추가한다.

```
okPushButton = new QPushButton( this, "ok button" );
okPushButton->setText( "OK" );
okPushButton->setDefault( TRUE );
buttonBox->addWidget( okPushButton );
cancelPushButton = new QPushButton( this, "cancel button" );
cancelPushButton->setText( "Cancel" );
cancelPushButton->setAccel( Key_Escape );
buttonBox->addWidget( cancelPushButton );
```

OK와 Cancel단추를 창조하여 buttonBox에 추가하였다. OK단추를 대화칸의 기정단추로 만들고 Esc건을 Cancel단추의 지름건으로 만든다.

```
tableButtonBox->addLayout( buttonBox );
```

buttonBox배치관리자를 tableButtonBox에 추가하고 배치를 완성한다.

```
connect( table, SIGNAL( clicked(int,int,int,const QPoint&) ),
   this, SLOT( setColor(int,int) ) );
connect( table, SIGNAL( currentChanged(int,int) ),
   this, SLOT( currentChanged(int,int) ) );
connect( table, SIGNAL( valueChanged(int,int) ),
   this, SLOT( valueChanged(int,int) ) );
connect( colorPushButton, SIGNAL( clicked() ), this, SLOT( setColor() ) );
connect( okPushButton, SIGNAL( clicked() ), this, SLOT( accept() ) );
connect( cancelPushButton, SIGNAL( clicked() ), this, SLOT( reject() ) );
```

이제는 폼을 《련결(wire up)》한다.

- 사용자가 세포를 하나 찰칵하면 setColor()처리부를 호출한다. 이것은 세포가 색을 보유하고 있으면 색대화칸을 펼친다.
- QTable의 currentChanged()신호를 자체의 currentChanged()처리부에 련결한다. 이것은 색단추를 (실례로 사용자가 어느 란안에 있는가에 따라서) 허용/금지하는데 쓰인다.

• 표의 valueChanged()신호를 자체의 valueChanged()처리부에 련결한다. 이것을 사용하여 값을 정확한 개수의 10진자리로 현시한다.

• 사용자가 Color단추를 찰칵하면 setColor()처리부를 호출한다.

• OK단추는 accept()처리부에 련결된다. 이 처리부에서 요소벡토르를 갱신한다.

• Cancel단추는 QDialog의 reject()처리부에 련결되고 코드와 작용을 더는 요구하지 않는다.

```
QPixmap patterns[MAX_PATTERNS];
patterns[0]  = QPixmap( pattern01 );
patterns[1]  = QPixmap( pattern02 );
```

매개 솔패턴의 픽스매프를 창조하여 patterns 배렬에 보관한다.

```
QRect rect = table->cellRect( 0, 1 );
QPixmap pix( rect.width(), rect.height() );
```

매개 색세포가 차지하는 직4각형을 얻고 그 크기의 빈 픽스매프를 창조한다.

```
for ( int i = 0; i < ChartForm::MAX_ELEMENTS; ++i ) {
 Element element = (*m_elements)[i];

 if ( element.isValid() )
  table->setText( i, 0, QString( "%1" ).arg( element.value(), 0, 'f',
m_decimalPlaces ) );
 QColor color = element.valueColor();
 pix.fill( color );
 table->setPixmap( i, 1, pix );
 table->setText( i, 1, color.name() );
 QComboBox *combobox = new QComboBox;
 for ( int j = 0; j < MAX_PATTERNS; ++j )
  combobox->insertItem( patterns[j] );
 combobox->setCurrentItem( element.valuePattern() - 1 );
 table->setCellWidget( i, 2, combobox );
 table->setText( i, 3, element.label() );
 color = element.labelColor();
 pix.fill( color );
 table->setPixmap( i, 4, pix );
 table->setText( i, 4, color.name() );
```

요소벡토르의 매개 요소들로 표를 채워넣어야 한다.

요소가 유효이면 특정한 개수의 10진자리수로 형식화하여 첫 란에 그 값을 써넣는다.

요소값의 색을 읽어들이고 빈 픽스매프를 그 색으로 채운다. 그다음 현시하려는 색세포를 이 픽스매프로 설정한다. 후에 반대로 색을 읽어들일수 있어야 한다(실례로 사용자가 그것을 변경한다면). 이것을 수행하는 한가지 방법은 픽스매프안의 화소를 시험하는것이고 다른 방법은 (CanvasText파생클라스에서와 같은 방법으로) QTableItem의 파생클라스를 만들고 거기에 색을 보관하는것이다. 그러나 더 간단한 로정을 선택하고 세포의 본문을 색이름으로 설정한다.

다음에 패턴복합칸을 패턴들로 채운다. 복합칸에서 선택한 패턴의 위치를 리용하여 사용자가 선택한 패턴을 결정한다. QTable은 QComboTableItem항목들을 사용할수 있으나 이것들은 본문만 보유하므로 setCellWidget()을 사용하여 표에 QComboBox의 항목들을 삽입한다.

다음에 요소의 표식자를 삽입한다. 끝으로 값의 색을 설정할 때와 같은 방법으로

표식자색을 설정한다.

(2) 처리부

```
void SetDataForm::currentChanged( int, int col )
{
 colorPushButton->setEnabled( col == 1 || col == 4 );
}
```

사용자가 표를 항행할 때 currentChanged()신호가 발생된다. 사용자가 1렬이나 4렬(값의 색 혹은 표식자색)에 입력할 때 colorPushButton을 허용하고 그렇지 않으면 그것을 금지한다.

```
void SetDataForm::valueChanged( int row, int col )
{
 if ( col == 0 ) {
  bool ok;
  double d = table->text( row, col ).toDouble( &ok );
  if ( ok && d > EPSILON )
   table->setText( row, col, QString( "%1" ).arg( d, 0, 'f',
m_decimalPlaces ) );
  else if ( !table->text( row, col ).isEmpty() )
   table->setText( row, col, table->text( row, col ) + "?" );
 }
}
```

사용자가 값을 변경하면 정확한 개수의 10진자리들로 그것을 형식화하거나 그것이 무효라는것을 지적해야 한다.

```
void SetDataForm::setColor()
{
 setColor( table->currentRow(), table->currentColumn() );
 table->setFocus();
}
```

사용자가 Color단추를 찰칵하면 다른 setColor()함수를 호출하고 표에로 초점을 돌려야 한다.

```
void SetDataForm::setColor( int row, int col )
{
 if ( !( col == 1 || col == 4 ) )
  return;
 QColor color = QColorDialog::getColor( QColor( table->text( row,
                         col ) ), this, "color dialog" );
 if ( color.isValid() ) {
  QPixmap pix = table->pixmap( row, col );
  pix.fill( color );
  table->setPixmap( row, col, pix );
  table->setText( row, col, color.name() );
 }
}
```

이 함수가 색세포에 초점이 놓이여 호출되면 정적 QColorDialog::getColor()대화칸을 호출하여 사용자의 색선택을 얻는다. 색을 선택하면 색세포의 픽스매프를 그 색으로 채우고 세포의 본문을 새로운 색의 이름으로 설정한다.

```
void SetDataForm::accept()
{
 bool ok;
 for ( int i = 0; i < ChartForm::MAX_ELEMENTS; ++i ) {
  Element &element = (*m_elements)[i];
  double d = table->text( i, 0 ).toDouble( &ok );
  if ( ok )
   element.setValue( d );
  else
   element.setValue( Element::INVALID );
  element.setValueColor( QColor( table->text( i, 1 ) ) );
  element.setValuePattern(((QComboBox*)table->cellWidget( i, 2 ))-
                      >currentItem()+1);
  element.setLabel( table->text( i, 3 ) );
  element.setLabelColor( QColor( table->text( i, 4 ) ) );
 }

 QDialog::accept();
}
```

사용자가 OK를 찰칵하면 요소벡토르를 갱신한다. 벡토르를 순환하면서 매개 요소의 값을 사용자가 입력한 값으로 설정하고 값이 무효이면 INVALID로 설정한다. 인수와 같은 색이름을 가지는 QColor를 일시 구성함으로써 값의 색과 표식자색을 설정한다. 패턴은 번호 1을 가지는 패턴복합칸의 현재항목으로 설정된다. (패턴번호는 1로 시작하지만 복합칸의 항목들은 첨수 0으로 시작한다.)

끝으로 QDialog::accept()를 호출한다.

8. 환경의 설정

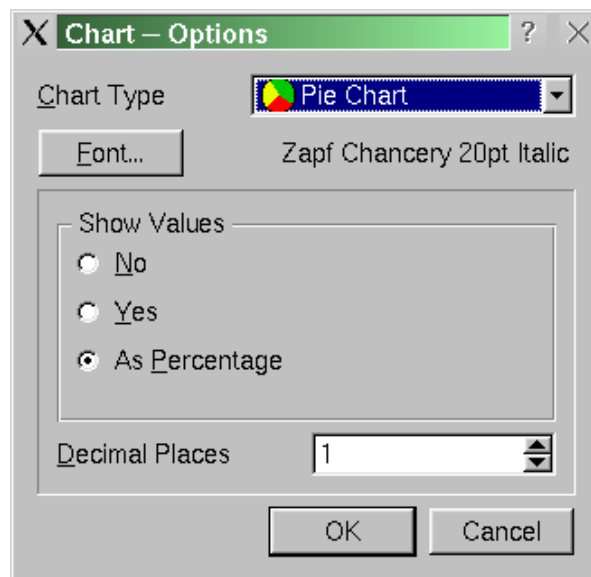



그림 2-19. 환경의 설정

선택대화칸을 제공하여 사용자가 한곳에서 모든 자료모임들에 적용하는 환경선택값들을 설정하도록 한다.

```
// optionsform.h로부터 발취
class OptionsForm : public QDialog
{
 Q_OBJECT
public:
 OptionsForm( QWiget* parent = 0, const char* name = "options form",
     bool modal = FALSE, WFlags f = 0 );
 ~OptionsForm() {}

 QFont font() const { return m_font; }
 void setFont( QFont font );

 QLabel *chartTypeTextLabel;
 QComboBox *chartTypeComboBox;
 QPushButton *fontPushButton;
 QLabel *fontTextLabel;
 QFrame *addValuesFrame;
 QButtonGroup *addValuesButtonGroup;
 QRadioButton *noRadioButton;
 QRadioButton *yesRadioButton;
 QRadioButton *asPercentageRadioButton;
 QLabel *decimalPlacesTextLabel;
 QSpinBox *decimalPlacesSpinBox;
 QPushButton *okPushButton;
 QPushButton *cancelPushButton;

protected slots:
 void chooseFont();

protected:
 QVBoxLayout *optionsFormLayout;
 QHBoxLayout *chartTypeLayout;
 QHBoxLayout *fontLayout;
 QVBoxLayout *addValuesFrameLayout;
 QVBoxLayout *addValuesButtonGroupLayout;
 QHBoxLayout *decimalPlacesLayout;
 QHBoxLayout *buttonsLayout;

private:
 QFont m_font;
};
```

이 대화칸의 배치는 자료설정폼보다 훨씬 더 복잡하지만 하나의 처리부만 요구한다. 자료설정폼과 달리 이것은 호출자가 설정하고 읽어들이는 창문부품들을 단순히 제공하는 대화칸이다. 호출자는 사용자가 만드는 변경에 기초한 갱신에 대응할수 있다.

```
// optionsform.cpp로부터 발취
 #include "images/options_horizontalbarchart.xpm"
 #include "images/options_piechart.xpm"
 #include "images/options_verticalbarchart.xpm"
```

도표형복합칸에 사용하려는 픽스매프들을 포함한다.

(1) 구성자

```
 OptionsForm::OptionsForm( QWiget* parent, const char* name, bool modal, WFlags f )
   : QDialog( parent, name, modal, f )
 {
   setCaption( "Chart -- Options" );
   resize( 320, 290 );
```

QDialog구성자에 인수들을 모두 넘기여 제목과 초기크기를 설정한다.

폼의 배치는 수평칸배치관리자에 도표형표식자과 복합칸을 가지며 서체단추와 서체표식자, 그리고 10진자리수표식자과 스핀칸에서도 비슷하다. 단추들도 역시 수평배치관리자에 배치되지만 그것들을 오른쪽으로 이동하기 위한 수축자가 있다. 표시값 라지오단추들은 틀의 수직칸배치관리자에 배치된다.

```
   optionsFormLayout = new QVBoxLayout( this, 11, 6 );
```

모든 창문부품들은 폼의 수직칸배치관리자안에 배치된다.

```
   chartTypeLayout = new QHBoxLayout( 0, 0, 6 );
```

도표형표식자과 복합칸은 나란히 배치된다.

```
   chartTypeTextLabel = new QLabel( "&Chart Type", this );
   chartTypeLayout->addWiget ( chartTypeTextLabel );

   chartTypeComboBox = new QComboBox( FALSE, this );
   chartTypeComboBox->insertItem( QPixmap( options_piechart ), "Pie Chart" );
   chartTypeComboBox->insertItem( QPixmap( options_verticalbarchart ),
    "Vertical Bar Chart" );
   chartTypeComboBox-insertItem( QPixmap( options_horizontalbarchart ),
    "Horizontal Bar Chart" );
   chartTypeLayout->addWiget( chartTypeComboBox );
   optionsFormLayout->addLayout( chartTypeLayout );
```

도표형표식자를 창조한다. (후에 도표형복합칸과 관련한 지름건을 준다.) 또한 도표형복합칸을 창조하고 거기에 픽스매프들과 본문을 둘다 채워넣는다. 그것들을 둘다 수평배치관리자에 추가하고 그 수평배치관리자를 폼의 수직배치관리자에 추가한다.

```
   fontLayout = new QHBoxLayout( 0, 0, 6 );

   fontPushButton = new QPushButton( "&Font...", this );
   fontLayout->addWiget( fontPushButton );
   QSpacerItem* spacer = new QSpacerItem( 0, 0,
                      QSizePolicy::Expanding,QSizePolicy::Minimum );
   fontLayout->addItem( spacer );
   fontTextLabel = new QLabel( this ); // Must be set by caller via setFont()
   fontLayout->addWiget( fontTextLabel );
   optionsFormLayout->addLayout( fontLayout );
```

서체단추와 서체표식자를 보유하기 위한 수평칸배치관리자를 창조한다. 서체단추는

단순하다. 수축자를 추가하여 겉모양을 개선한다. 현재 사용자가 사용할 서체를 모르므로 서체본문표식자는 처음에 비여있다.

```
addValuesFrame = new QFrame( this );
addValuesFrame->setFrameShape( QFrame::StyledPanel );
addValuesFrame->setFrameShadow( QFrame::Sunken );
addValuesFrameLayout = new QVBoxLayout( addValuesFrame, 11, 6 );
addValuesButtonGroup = new QButtonGroup( "Show Values", addValuesFrame );
addValuesButtonGroup->setColumnLayout(0, Qt::Vertical );
addValuesButtonGroup->layout()->setSpacing( 6 );
addValuesButtonGroup->layout()->setMargin( 11 );
addValuesButtonGroupLayout = new QVBroxLayout(
addValuesButtonGroup->layout() );
addValuesButtonGroupLayout->setAlignment( Qt::AlignTop );

noRadioButton = new QRadioButton( "&No", addValuesButtonGroup );
noRadioButton->setChecked( TRUE );
addValuesButtonGroupLayout->addWiget ( noRadioButton );

yesRadioButton = new QRadioButton( "&Yes", addValuesButtonGroup );
addValuesButtonGroupLayout->addWiget ( yesRadioButton );

asPercentageRadioButton = new QRadioButton( "As &Percentage",
  addValuesButtonGroup );
addValuesButtonGroupLayout->addWiget( asPercentageRadioButton );
addValuesFrameLayout->add창문부품( addValuesButtonGroup );
```

사용자는 표식자들을 현시하거나 매개 표식자의 옆에 값들을 그대로 혹은 퍼센트로 추가하도록 선택할수 있다.

틀을 창조하고 그 안에 라지오단추들을 표시하고 그것들의 배치관리자를 창조한다. 단추그룹을 창조하여 Qt가 배타적인 라지오단추의 동작을 자동적으로 조종하게 한다. 다음에 라지오단추들을 창조한다. 이것들은 기정으로 No를 가진다.

10진자리수표식자과 스핀칸은 다른 수평배치관리자들처럼 배치되고 단추들은 자료설정폼의 단추들과 아주 류사한 방법으로 배치된다.

```
connect( fontPushButton, SIGNAL( clicked() ), this,
         SLOT( chooseFont() ) );
connect( okPushButton, SIGNAL( clicked() ), this,
         SLOT( accept() ) );
connect( cancelPushButton, SIGNAL( clicked() ), this,
         SLOT( reject() ) );
```

오직 3개의 련결만 필요하다.

① 사용자가 서체단추를 찰칵할 때 자체의 chooseFont()처리부를 실행한다.

② 사용자가 OK를 찰칵할 때 QDialog::accept()를 호출한다. 이것은 호출자가 대화칸의 창문부품들로부터 자료를 읽어들여 필요한 작용들을 수행하게 한다.

③ 사용자가 Cancel을 찰칵하면 QDialog::reject()를 호출한다.

```
chartTypeTextLabel->setBuddy( chartTypeComboBox );
```

```
decimalPlacesTextLabel->setBuddy( decimalPlacesSpinBox );
```

setBuddy()함수를 사용하여 창문부품들을 표식자지름건들과 련결한다.

(2) 처리부

```
void OptionsForm::chooseFont()
{
 bool ok;
 QFont font = QFontDialog::getFont( &ok, m_font, this );
 if ( ok )
  setFont( font );
}
```

사용자가 Font단추를 찰칵할 때 이 처리부가 호출된다. 이것은 단지 정적함수 QFontDialog::getFont()를 호출하여 사용자의 서체선택을 얻는다. 서체를 선택하였다면 서체표식자에 서체의 본문서술을 제시하는 setFont() 처리부를 호출한다.

```
void OptionsForm::setFont( QFont font )
{
 QString label = font.family() + " " +
     QString::number( font.pointSize() ) + "pt";
 if ( font.bold() )
  label += " Bold";
 if ( font.italic() )
  label += " Italic";
 fontTextLabel->setText( label );
 m_font = font;
}
```

이 함수는 선택한 서체의 본문서술을 서체표식자에 현시하고 m_font성원에 서체의 사본을 보관한다. chooseFont()에 기정서체를 제공하기 위하여 서체를 성원으로 요구한다.

9. 프로젝트파일

```
// chart.pro
 TEMPLATE = app
 CONFIG  += warn_on
 REQUIRES = full-config
 HEADERS += element.h \
     canvastext.h \
     canvasview.h \
     chartform.h \
     optionsform.h \
     setdataform.h
 SOURCES += element.cpp \
     canvasview.cpp \
     chartform.cpp \
     chartform_canvas.cpp \
     chartform_files.cpp \
     optionsform.cpp \
     setdataform.cpp \
```

main.cpp
qmake를 실행하여 Makefile을 생성한다.
qmake -o Makefile chart.pro

chart프로그람은 Qt에 의하여 응용프로그람과 대화칸을 창조하는 간단한 방법을 보여준다. 차림표와 도구띠의 창조는 간단하고 Qt의 신호-처리부기구는 GUI사건조종을 상당히 단순화한다.

수동적인 배치관리자창조를 배울 기회가 있을수 있으나 간단한 방법은 QtDesigner이다. QtDesigner는 단순하지만 강력한 배치도구와 코드편집기를 포함하고있으며 자동적으로 main.cpp와 .pro 프로젝트파일을 창조한다.

앞으로 chart응용프로그람에 다음의 기능을 실현할수 있다.

- QValidator파생클라스를 사용하여 오직 유효한 double들만 값으로 입력될수 있도록 담보한다.
- 더 많은 도표형 즉 직선그라프, 령역그라프와 힐로(hi-lo)그라프의 추가.
- 사용자가 우, 아래, 오른쪽, 왼쪽여백을 설정하게 한다.
- 사용자가 표식자과 같이 끌기할수 있는 항목을 지정한다.
- 선택항목을 그리고 표식하는 축의 제공.
- 표식자대신에 건을 제공하기 위한 선택의 제공.
- 모든 도표형들에 3D형식선택의 추가.

제4절. 간단한 응용프로그람

이 절에서는 현대응용프로그람창문에서 사용하고있는 클라스들인 QMainWindow, QMenuBar, QPopupMenu, QToolBar 및 QStatusBar의 간단한 사용을 보여준다.

또한 QWhatsThis(간단한 방조용)의 몇가지 국면과 QApplication을 사용하는 전형적인 main()을 설명한다.

끝으로 QPrinter에 기초한 전형적인 인쇄함수를 보여준다.

1. ApplicationWindow의 선언

여기에 완전한 머리부파일이 있다.

```
#ifndef APPLICATION_H
#define APPLICATION_H

#include <qmainwindow.h>
class QTextEdit;

class ApplicationWindow: public QMainWindow
{
 Q_OBJECT

public:
 ApplicationWindow();
 ~ApplicationWindow();

protected:
 void closeEvent( QCloseEvent* );
```

```
private slots:
 void newDoc();
 void choose();
 void load( const QString &fileName );
 void save();
 void saveAs();
 void print();

 void about();
 void aboutQt();

private:
 QPrinter *printer;
 QTextEdit *e;
 QString filename;
};

#endif
```

이것은 QMainWindow을 계승하며 처리부와 비공개변수들을 가지는 클라스를 선언한다. 선두에서 QTextEdit의 클라스앞방향선언(머리부대신에)은 콤파일속도를 높인다. 이러한 방법으로 make depend는 qtextedit.h가 변경될 때 application.h을 포함하는 매개의 .cpp파일의 재콤파일을 요구하지 않는다.

2. 간단한 main()

여기에 안전한 main.cpp이 있다.

```
#include <qapplication.h>
#include "application.h"

int main( int argc, char ** argv ) {
 QApplication a( argc, argv );
 ApplicationWindow *mw = new ApplicationWindow();
 mw->setCaption( "Qt Example - Application" );
 mw->show();
 a.connect( &a, SIGNAL(lastWindowClosed()), &a, SLOT(quit()) );
 return a.exec();
}
```

그러면 main.cpp을 구체적으로 고찰하자.

```
 int main( int argc, char ** argv ) {
  QApplication a( argc, argv );
```

우의 행에서 보통의 구성자를 가지는 QApplication객체를 창조하고 argc와 argv를 해석한다. QApplication자체는 -geometry와 같은 X11에 고유한 지령행추가선택을 고려하므로 프로그람은 자동적으로 X의뢰기들이 바라는대로 동작한다.

```
  ApplicationWindow *mw = new ApplicationWindow();
  mw->setCaption( "Qt Example - Application" );
  mw->show();
```

제일웃준위창문부품으로서 ApplicationWindow를 창조하고 창문제목을

"Document 1"로 설정하고 창문을 표시한다.

```
a.connect( &a, SIGNAL(lastWindowClosed()), &a, SLOT(quit()) );
```

응용프로그람의 마지막 창문이 닫길 때 완료해야 한다. 신호와 처리부들은 둘다 QApplication의 사전정의된 성원들이다.

```
return a.exec();
```

응용프로그람의 초기화를 완성하고 기본사건순환고리(GUI)를 기동하며 우연히 사건순환고리를 떠날 때 QApplication의 오유코드를 돌려준다.

```
}
```

3. ApplicationWindow의 실현

실현파일이 아주 크므로(약 300행) 완전히 렬거하지 않는다. (원천코드는 examples/application등록부에 포함된다.) 구성자로 기동하기전에 언급할만한 가치가 있는 3개의 #include가 있다.

```
#include "filesave.xpm"
#include "fileopen.xpm"
#include "fileprint.xpm"
```

응용프로그람의 도구단추들은 그림기호들이 없으면 제대로 표시되지 않는다. 그림기호들은 우에서 포함한 XPM파일들에 있다.

```
ApplicationWindow::ApplicationWindow()
  : QMainWindow( 0, "example application main window",
      WDestructiveClose | WGroupLeader )
{
```

ApplicationWindow는 차림표띠, 도구띠 등을 가지는 전형적인 응용프로그람기본창문을 제공하는 Qt클라스인 QMainWindow를 계승한다.

```
printer = new QPrinter( QPrinter::HighResolution );
```

응용프로그람실례는 무엇인가 인쇄하고 인쇄시에 사용자가 설정값을 변경할 때 새로운 설정이 다음번에 기정으로 되도록 QPrinter객체를 선택한다.

```
QPixmap openIcon, saveIcon, printIcon;
```

실레는 도구띠에 간단히 여러개의 지령들을 가진다. 우의 변수들은 그 매개에 대하여 그림기호를 하나씩 보유한다.

```
QToolBar * fileTools = new QToolBar( this, "file operations" );
```

이 창문안에 도구띠를 창조하고

```
fileTools->setLabel( "File Operations" );
```

제목을 정의한다. 사용자가 도구띠를 그 위치밖으로 끌고가서 탁상우에 띄여놓을 때 도구띠창문은 제목으로서 "File Operations"를 표시한다.

```
openIcon = QPixmap( fileopen );
QToolButton * fileOpen
  = new QToolButton( openIcon, "Open File", QString::null,
        this, SLOT(choose()), fileTools, "open file" );
```

이제 적당한 그림기호와 도구암시본문 "Open File"을 가지는 fileTools 도구띠용의 첫 도구단추를 창조한다. 선두에 포함한 fileopen.xpm 은 fileopen라고 부르는 픽스매프의 정의를 포함한다. 이 그림기호를 사용하여 첫 도구단추를 설명한다.

```
saveIcon = QPixmap( filesave );
QToolButton * fileSave = new QToolButton( saveIcon, "Save File",
          QString::null,this, SLOT(save()), fileTools, "save file" );
printIcon = QPixmap( fileprint );
```

```
QToolButton * filePrint = new QToolButton( printIcon, "Print File",
            QString::null,this, SLOT(print()), fileTools, "print file" );
```

비슷한 방법으로 도구띠에 두개의 도구단추를 더 창조한다. 매개가 적당한 그림기호들과 도구암시본문을 가진다. 3개의 단추는 모두 이 객체의 적당한 처리부들에 련결된다. 실례로 "Print File"단추는 ApplicationWindow::print()에 련결된다.

```
(void)QWhatsThis::whatsThisButton( fileTools );
```

도구띠의 4번째 단추는 "What's This?"방조를 제공한다. 그 마우스대면부가 보통과 다르므로 특수한 함수를 리용하여 설정되여야 한다.

```
const char * fileOpenText = "<p><img source=\"fileopen\"> "
    "Click this button to open a <em>new file</em>.<br>"
    "You can also select the <b>Open</b> command "
    "from the <b>File</b> menu.</p>";

QWhatsThis::add( fileOpen, fileOpenText );
```

우의 행에서 fileOpen 단추에 "What's This?"방조본문을 추가하고

```
QMimeSourceFactory::defaultFactory()->setPixmap( "fileopen", openIcon );
```

방조본문(fileOpenText에 보관한것처럼)이 "fileopen"이라는 이름의 화상을 요구할 때 openIcon 픽스매프가 사용된다는것을 리치본문엔진에 알린다.

```
const char * fileSaveText = "<p>Click this button to save the file you "
    "are editing. You will be prompted for a file name.\n"
    "You can also select the <b>Save</b> command "
    "from the <b>File</b> menu.</p>";

QWhatsThis::add( fileSave, fileSaveText );
const char * filePrintText = "Click this button to print the file you
are editing.\n" "You can also select the Print command from the File
menu.";

QWhatsThis::add( filePrint, filePrintText );
```

나머지 두개 단추의 "What's This?"방조는 픽스매프를 사용하지 않으므로 단추에 방조본문을 추가하여야 한다.

주의: fileSaveText()에서 리치본문요소들을 호출하기 위하여 전체문자렬은 <p>와 </p>에의해 둘러막아야 한다. filePrintText()는 리치본문요소들을 가지지 않으므로 이것은 필요없다.

```
QPopupMenu * file = new QPopupMenu( this );
menuBar()->insertItem( "&File", file );
```

다음으로 File차림표용의 QPopupMenu을 창조하여 차림표띠에 추가한다. 문자 F앞에 &를 붙이면 사용자가 지름건 Alt+F 을 사용하여 차림표를 펼칠수 있다.

```
file->insertItem( "&New", this, SLOT(newDoc()), CTRL+Key_N );
```

그 첫 항목이 처리부 newDoc()에 련결된다.(아직 실현되지 않는다.) 사용자가 이 New 항목을 선택하거나 (실례로 &표식된 문자N을 입력하여) Ctrl+N 지름건을 사용하면 새로운 편집기창문이 펼쳐진다.

```
int id;
id = file->insertItem( openIcon, "&Open...", this, SLOT(choose()),
CTRL+Key_O );
file->setWhatsThis( id, fileOpenText );
```

```
id = file->insertItem( saveIcon, "&Save", this, SLOT(save()), CTRL+Key_S );
file->setWhatsThis( id, fileSaveText );

id = file->insertItem( "Save &As...", this, SLOT(saveAs()) );
file->setWhatsThis( id, fileSaveText );
```
File차림표를 3개의 지령들(Open, Save 그리고 Save As)로 채우고 그것들에 대하여 "What's This?"방조를 설정한다. 특히 "What's This?"방조와 픽스매프들은 도구띠와 차림표띠에서 둘다 사용된다. (QAction과 《 Qt실레프로그람 》의 examples/action실레를 참고.)
```
file->insertSeparator();
```
그다음 분리선을 삽입하고
```
id = file->insertItem( printIcon, "&Print...", this, SLOT(print()), CTRL+Key_P );
file->setWhatsThis( id, filePrintText );

file->insertSeparator();
file->insertItem( "&Close", this, SLOT(close()), CTRL+Key_W );
file->insertItem( "&Quit", qApp, SLOT( closeAllWindows() ), CTRL+Key_Q );
```
"What's This?"방조를 가지는 Print지령, 또 하나의 분리선, "What's This?"와 픽스매프들이 없는 2개의 지령을 삽입한다. Close지령의 경우에 신호는 각각의 ApplicationWindow객체의 close() 처리부에 련결되고 Quit 지령은 전체 응용프로그람에 영향을 준다.

ApplicationWindow는 QWidget이므로 close() 함수는 후에 실현하는 closeEvent()에로의 호출로 절환한다.
```
menuBar()->insertSeparator();
```
File차림표를 실현하였으므로 초점을 차림표띠로 옮기고 분리선을 삽입한다. 이제부터 앞으로 차림표띠항목들은 창문체계형식이 요구한다면 오른쪽에 배치된다.
```
QPopupMenu * help = new QPopupMenu( this );
menuBar()->insertItem( "&Help", help );

help->insertItem( "&About", this, SLOT(about()), Key_F1 );
help->insertItem( "About &Qt", this, SLOT(aboutQt()) );
help->insertSeparator();
help->insertItem( "What's &This", this, SLOT(whatsThis()), SHIFT+Key_F1 );
```
Help차림표를 창조하여 차림표띠에 추가하고 여러개의 지령을 삽입한다. 형식에 따라서 지령들은 차림표띠의 오른쪽에 나타난다.
```
e = new QTextEdit( this, "editor" );
e->setFocus();
setCentralWidget( e );
```
이제 하나의 본문편집기를 창조하고 처음의 초점을 그것에 설정하고 창문의 중심창문부품으로 만든다.

QMainWindow::centralWidget()는 전체 응용프로그람의 심장부이다. 거기에 차

림표띠, 상태띠, 도구띠들이 모두 배렬된다. 중심창문부품이 본문편집할수 있는 창문부품이므로 우리의 응용프로그람은 간단한 본문편집기이다.

```
statusBar()->message( "Ready", 2000 );
```

상태띠가 기동시에 2s동안 Ready를 표시하여 사용자에게 그 창문이 초기화를 끝냈으며 사용가능하다는것을 알리게 한다.

```
resize( 450, 600 );
```

끝으로 새로운 창문의 크기를 기정크기로 조절한다.

```
}
```

구성자를 끝내고 해체자를 설명한다.

```
ApplicationWindow::~ApplicationWindow()
{
 delete printer;
}
```

ApplicationWindow창문부품이 해체자에서 해야 할 유일한 일은 그것이 창조한 인쇄기를 삭제하는것이다. 다른 모든 객체들은 적당한 시각에 Qt가 삭제하는 자식창문부품들을 가진다.

이제는 머리부파일에서 언급하고 구성자에서 사용한 처리부들을 모두 실현하는것이다.

```
void ApplicationWindow::newDoc()
{
 ApplicationWindow *ed = new ApplicationWindow;
 ed->setCaption("Qt Example - Application");
 ed->show();
}
```

File|New차림표항목에 련결된 이 처리부는 단순히 새로운 ApplicationWindow를 창조하고 표시한다.

```
void ApplicationWindow::choose()
{
 QString    fn    =    QFileDialog::getOpenFileName(    QString::null,
QString::null, this);
 if ( !fn.isEmpty() )
  load( fn );
 else
  statusBar()->message( "Loading aborted", 2000 );
}
```

choose()처리부는 Open차림표항목과 도구단추에 련결된다. QFileDialog::getOpenFileName()로부터의 약간한 방조에 의하여 사용자에게 파일이름을 묻고 그다음 그 파일을 적재하거나 상태띠에 오유통보를 준다.

```
void ApplicationWindow::load( const QString &fileName )
{
 QFile f( fileName );
 if ( !f.open( IO_ReadOnly ) )
  return;

 QTextStream ts( &f );
 e->setText( ts.read() );
 e->setModified( FALSE );
```

```
  setCaption( fileName );
  statusBar()->message( "Loaded document " + fileName, 2000 );
 }
```

이 함수는 편집기에 파일을 적재한다. 그것을 수행하면 창문체계제목을 파일이름으로 설정하고 상태띠에 약 2s동안 성공통보를 현시한다. 존재하지 않지만 읽을수 없는 파일들에 의하여 어떤 일도 생기지 않는다.

```
 void ApplicationWindow::save()
 {
  if ( filename.isEmpty() ) {
   saveAs();
   return;
  }

  QString text = e->text();
  QFile f( filename );
  if ( !f.open( IO_WriteOnly ) ) {
   statusBar()->message( QString("Could not write to %1").arg(filename),
         2000 );
   return;
  }

  QTextStream t( &f );
  t << text;
  f.close();
```

이름을 제안할 때 함수는 현재 파일을 보관한다. 지금까지 파일이름이 지정되지 않았으면 saveAs()함수가 호출된다. 써넣을수 없는 파일들은 ApplicationWindow객체가 상태띠에 오유통보를 제공하게 한다. 이것을 수행하는 방법이 하나이상 있다. 우의 statusBar()->message()행을 load()함수의 등가한 코드와 비교할수 있다.

```
  e->setModified( FALSE );
```

편집기에서 마지막으로 보관한 다음 내용을 편집하지 않았다는것을 알린다. 사용자가 더 편집하고 명백히 보관하지 않고 창문을 닫으려고 할 때 ApplicationWindow::closeEvent()가 보관하겠는가 묻는다.

```
  setCaption( filename );
```

문서가 낡은 제목이 아닌 다른 이름으로 보관될수 있으므로 창문제목을 확인하도록 설정한다.

```
  statusBar()->message( QString( "File %1 saved" ).arg( filename ), 2000 );
 }
```

상태띠의 통보를 리용하여 사용자에게 그 파일이 성공적으로 보관되였다는것을 통보한다.

```
 void ApplicationWindow::saveAs()
 {
  QString fn = QFileDialog::getSaveFileName( QString::null,
QString::null, this );
  if ( !fn.isEmpty() ) {
   filename = fn;
   save();
```

```
    } else {
     statusBar()->message( "Saving aborted", 2000 );
    }
   }
```

이 함수는 새 이름을 묻고 그 이름으로 문서를 보관하고 창문체계제목을 새 이름으로 암시적으로 변경한다.

```
void ApplicationWindow::print()
{
  printer->setFullPage( TRUE );
  if ( printer->setup(this) ) {      // printer dialog
   statusBar()->message( "Printing..." );
   QPainter p;
   if( !p.begin( printer ) ) {      // paint on printer
    statusBar()->message( "Printing aborted", 2000 );
    return;
   }

   QPaintDeviceMetrics metrics( p.device() );
   int dpiy = metrics.logicalDpiY();
   int margin = (int) ( (2/2.54)*dpiy ); // 2 cm margins
   QRect view( margin, margin, metrics.width() - 2*margin,
metrics.height() - 2*margin );
   QSimpleRichText richText( QStyleSheet::convertFromPlainText(e-
>text()), QFont(),e->context(), e->styleSheet(), e->mimeSourceFactory(),
view.height() );
   richText.setWidth( &p, view.width() );
   int page = 1;
   do {
    richText.draw( &p, margin, margin, view, colorGroup() );
    view.moveBy( 0, view.height() );
    p.translate( 0 , -view.height() );
    p.drawText(                    view.right()                    -
p.fontMetrics().width( QString::number( page ) ),
        view.bottom()      +      p.fontMetrics().ascent()      +      5,
QString::number( page ) );
    if ( view.top() - margin >= richText.height() )
     break;
    printer->newPage();
    page++;
   } while (TRUE);

   statusBar()->message( "Printing completed", 2000 );
  } else {
   statusBar()->message( "Printing aborted", 2000 );
  }
}
```

print()는 File|Print차림표항목과 filePrint도구단추에 의해 호출된다.
사용자에게 인쇄설정대화칸을 현시하고 취소하면 인쇄를 무시한다.
QSimpleRichText객체를 창조하고 거기에 본문을 준다. 이 객체는 본문을 하나의 긴 페지로서 형식화할수 있다. QSimpleRichText페지로부터 한페지의 본문을 한번에 출력하여 페지번호를 얻는다.
이제는 사용자가 ApplicationWindow를 닫으려고 할 때 어떤 현상이 생기는가를 고찰한다.

```
void ApplicationWindow::closeEvent( QCloseEvent* ce )
{
```

이 사건은 창문체계닫기사건들을 처리하여 얻는다. 닫기사건은 은폐사건과 다르며 흔히 은페는 최소화를 의미하고 닫기는 창문이 사라지는것을 의미한다.

```
if ( !e->isModified() ) {
  ce->accept();
  return;
}
```

본문이 편집되지 않았으면 그 사건을 받아들인다. 창문은 닫겨지고 ApplicationWindow()구성자에서 WDestructiveClose창문부품기발을 사용하였으므로 창문부품이 삭제된다.

```
switch( QMessageBox::information( this, "Qt Application Example",
          "Do you want to save the changes to the document?",
          "Yes", "No", "Cancel", 0, 1 ) ) {
```

그렇지 않으면 사용자에게 무엇을 하려고 하는가고 묻는다.

```
case 0:
  save();
  ce->accept();
  break;
```

보관하고 완료하려고 한다면 그것을 수행한다.

```
case 1:
  ce->accept();
  break;
```

사용자가 완료하려고 하지 않으면 닫기사건을 무시한다. (그것을 차단할수 없는 경우가 있다.)

```
case 2:
default: // just for sanity
  ce->ignore();
  break;
```

마지막 경우 즉 사용자가 편집을 무시하고 완료하려고 하는 경우에는 아주 간단하다.

```
  }
}
```

끝으로 방조차림표항목들이 사용할 처리부들을 실현한다.

```
void ApplicationWindow::about()
{
 QMessageBox::about( this, "Qt Application Example",
      "This example demonstrates simple use of
      "QMainWindow,\nQMenuBar and QToolBar.");
}
```

```
void ApplicationWindow::aboutQt()
{
 QMessageBox::aboutQt( this, "Qt Application Example" );
}
```

이 두개의 처리부는 준비된 about함수들을 사용하여 이 프로그람과 그것이 사용하는 GUI도구일식에 대한 정보를 제공한다.

제3장. Qt객체모형

제1절. Qt객체모형

표준 C++객체모형은 객체원형을 위한 매우 효과적인 실행시지원기능을 제공한다. 그러나 C++객체모형의 정적특성은 일정한 문제령역에서 유연성이 없다. 도형방식사용자대면부프로그람작성은 실행시효률과 높은 수준의 유연성을 둘다 요구하는 령역이다. Qt는 Qt객체모형의 유연성과 함께 C++의 속도를 결합함으로써 이것을 제공한다.

Qt는 C++에 다음의 특성들을 추가한다.

- 신호, 처리부라고 부르는 원만한 객체통신을 위한 매우 강력한 기구
- 질문 및 설계가능한 객체속성들
- 강력한 사건들과 사건려과기들
- 국제화를 위한 문맥문자렬번역
- 사건구동형 GUI에서 많은 과제들을 훌륭히 통합할수 있게 하는 복잡한 시격구동형시계들
- 본래의 방법으로 객체소유를 조직하는 계층적이고 질문가능한 객체나무
- 감시(guarded)지적자 QGuardedPtr들은 참고된 객체가 해체되면 자동적으로 0으로 설정되지만 표준 C ++지적자들은 객체가 해체되면 《속박지적자(dangling pointer)》로 된다

Qt에서 이러한 대부분의 특징들은 QObject로부터 계승에 기초한 표준C++기술에 의해 실현된다. 객체통신기구와 동적속성체계와 같은 다른것들은 Qt자체의 메타객체콤파일러(moc)에 의해 제공되는 메타객체체계를 요구한다.

메타객체체계는 언어가 부분품GUI프로그람작성에 적합하게 한다. 형판을 C++확장에 사용할수 있지만 메타객체체계는 형판로 실현되지 않는다(6절 참고).

제2절. 객체나무와 객체소유자

QObject들은 그 자체를 객체나무로 조직화한다. 다른 객체를 부모로 하여 QObject를 창조할 때 그것은 부모의 children()목록에 추가되고 부모가 있을 때 삭제된다. 이 수법은 GUI객체들의 요구를 아주 충분히 만족시킨다. 실례로 QAccel(건반지름건)은 관련한 창문의 자식이므로 사용자가 창문을 닫을 때 지름건도 삭제된다.

정적함수 QObject::objectTrees()는 현재 존재하는 뿌리객체 모두에로의 접근을 제공한다.

화면에 나타나는 모든것의 기초클라스 QWidget는 부모-자식관계를 확장한다. 자식은 보통 자식창문부품으로도 된다. 즉 그것은 부모의 자리표계에 현시되고 부모의 경계에 의해 그라프적으로 갈라진다. 실례로 응용프로그람이 닫긴 후에 통보창을 삭제할 때 통보창의 단추와 표식자들도 삭제된다. 그것은 단추와 표식자들이 통보창의 자식이기때문이다.

또한 자식객체들을 자체로 삭제하면 부모들로부터 스스로 삭제된다. 실례로 사용자가 도구띠를 삭제할 때 응용프로그람이 QToolBar객체들중 하나를 삭제할수 있다. 그 경우에 도구띠의 QMainWindow부모는 변경을 탐지하고 그 화면공간을 재구성한다.

오유수정함수 QObject::dumpObjectTree()와 QObject::dumpObjectInfo()는 흔히 응용프로그람이 이상하게 보이거나 동작할 때 사용할수 있다.

제3절. 신호와 처리부

신호와 처리부는 객체들사이 통신에 사용된다. 신호-처리부기구는 Qt의 중요한 특징이며 다른 도구묶음(toolkit)과 가장 크게 차이나는 부분이다.

GUI프로그람작성에서는 자주 한 창문부품에서의 변화를 다른 창문부품에 통보할것을 요구한다. 더 일반적으로는 서로 통신할수 있는 임의의 종류의 객체들을 요구한다. 례들어 XML파일을 해석한다면 새 꼬리표를 만날 때마다 XML파일구조를 표시하는데 사용하고있는 목록보기에 통지하려고 한다.

낡은 도구묶음은 역호출에 의하여 이런 형태의 통신을 진행한다. 역호출은 함수의 지적자이므로 처리함수가 어떤 사건에 대해 통보하려고 하면 다른 함수의 지적자를 처리함수에 넘긴다. 처리함수는 그다음 적당한 역호출기능을 호출한다. 역호출은 두가지 기본결함이 있다. 첫째로, 형안전하지 못하다. 처리함수가 정확한 인수들을 가지고 역호출기능을 호출하는지 확신할수 없다. 둘째로, 처리함수는 어느 역호출이 호출되는지 알아야 하기때문에 역호출기능은 처리함수에 강하게 의존된다.

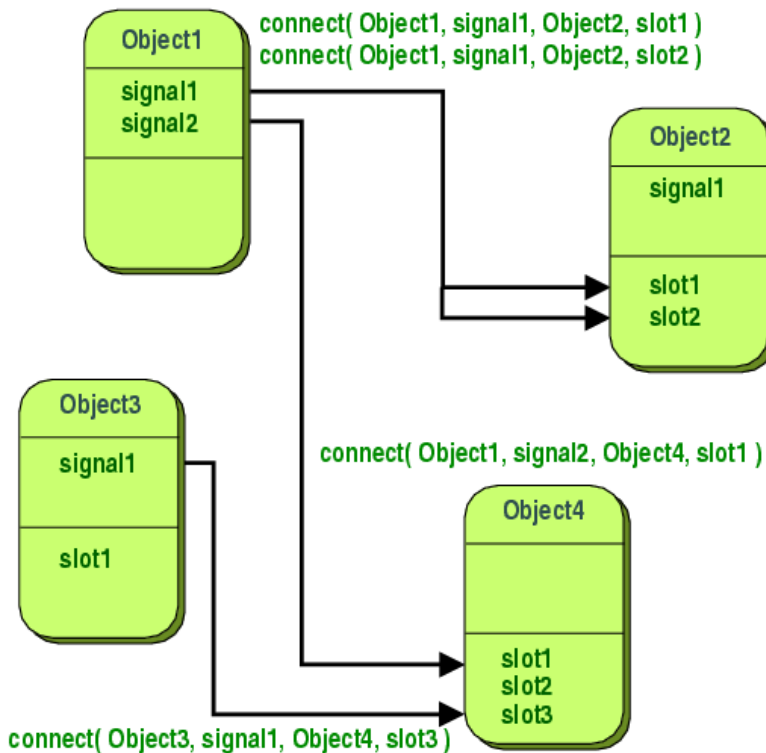



그림 3-1. 신호와 처리부련결의 추상화

Qt에서는 역호출기술대신 신호와 처리부를 사용한다. 신호는 특별한 사건이 발생하면 발생된다. Qt의 창문부품들은 미리 정의된 신호들을 많이 가지고 있지만 항상 파생클라스를 추가할수 있다. 처리부는 특별한 신호에 응답하여 호출되는 함수이다. Qt의 창문부품들은 미리 정의된 처리부를 많이 가지고있지만 실천에서는 일반적으로 자기가 관심을 가지는 신호들을 조종할수 있게 자체의 처리부들을 추가한다.

신호-처리부기구는 형안전하다. 신호의 서명은 받는측 처리부의 서명과 어울려야 한다. (사실상 처리부는 받는 신호보다 더 짧은 서명을 가진다. 왜냐하면 여유인수를 무시할수 있기때문이다.) 서명들은 서로 호환되므로 콤파일러가 형오유를 찾는데 도움이 될수 있다. 신호와 처리부는 약하게 결합된다. 신호를 발생하는 클라스는 어느 처리부가

신호를 받는지 모르며 관심하지도 않는다. Qt의 신호-처리부기구는 처리부에 신호를 련결(connect)하면 처리부가 적당한 시간내에 신호의 파라메터들을 가지고 호출된다는것을 담보한다. 신호와 처리부들은 임의의 형의 임의의 수의 인수를 가질수 있고 완전히 형안전하므로 더이상 역호출을 론의할 필요가 없다.

QObject 혹은 그 파생클라스들의 하나로부터 계승되는 모든 클라스(례들어 QWidget)들은 신호와 처리부를 포함할수 있다. 신호는 바깥세계에 관심을 가질수 있도록 객체들의 상태를 변화시킬 때 객체들에 의하여 발생된다. 이것은 객체들이 통신하게 한다. 발생하는 신호들을 어느것이 받는지 모르며 관심하지도 않는다. 이것은 진짜 정보은페이며 객체가 쏘프트웨어부분품으로 사용될수 있다는것을 담보한다.

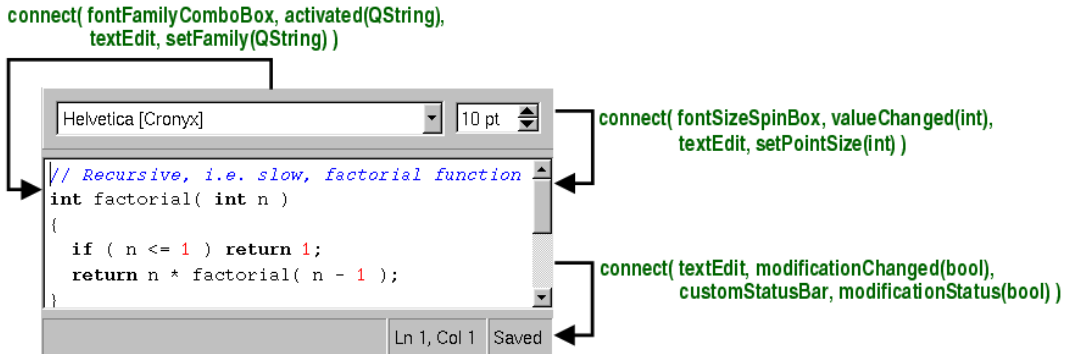



그림 3-2. 신호와 처리부련결의 실례

처리부는 받아들이는 신호에 사용될수 있는 보통성원함수들이다. 객체가 자기 신호를 어떤것이 받는지 모르는것처럼 처리부는 자기에게 어떤 신호가 련결되였는지 모른다. 이것은 Qt에 의하여 참말로 독립적인 부분품들을 창조할수 있게 한다.

한개의 처리부에 요구되는것만큼 많은 신호를 련결할수 있으며 한개 신호를 요구되는 수많은 처리부들에 련결할수 있다. 한개 신호를 다른 신호에 직접 련결하는것도 가능하다. (이것은 첫 신호가 발생될 때마다 두번째 신호가 즉시 발생된다.)

이와 함께 신호와 처리부는 강력한 부분품프로그람작성기술도 제공한다.

1. 간단한 실례

다음과 같이 아주 작은 C++클라스선언부를 쓴다.

```
class Foo
{
public:
  Foo();
  int value() const { return val; }
  void setValue( int );
private:
  int val;
};
```

작은 Qt클라스를 다음과 같이 쓴다.

```
class Foo : public QObject
{
  Q_OBJECT
public:
```

```
    Foo();
    int value() const { return val; }
  public slots:
    void setValue( int );
  signals:
    void valueChanged( int );
  private:
    int val;
  };
```

이 클라스는 같은 내부상태와 그리고 상태에 접근하는 공개메쏘드를 가진다. 그러나 그밖에도 신호와 처리부에 의한 부분품프로그람작성에 대한 지원도 가진다. 이 클라스는 신호 valueChanged()를 발생하여 상태가 변화되였다는것을 외부세계에 알릴수 있으며 다른 객체들이 신호들을 보낼수 있는 처리부를 가진다.

신호나 처리부를 가지는 모든 클라스는 자기 선언부에 Q_OBJECT를 서술해야 한다.

처리부는 응용프로그람작성자에 의해 실현된다. 여기에 Foo::setValue()의 가능한 실현부가 있다.

```
  void Foo::setValue( int v )
  {
    if ( v != val ) {
      val = v;
      emit valueChanged(v);
    }
  }
```

emit valueChanged(v)행은 객체로부터 신호 valueChanged를 발생시킨다. 알수 있는것처럼 emit signal(인수)를 리용하여 신호를 발생한다.

여기에 두개의 객체들을 서로 련결하는 방법이 있다:

```
  Foo a, b;
  connect(&a, SIGNAL(valueChanged(int)), &b, SLOT(setValue(int)));
  b.setValue( 11 ); // a == undefined  b == 11
  a.setValue( 79 ); // a == 79      b == 79
  b.value();    // returns 79
```

a.setValue(79)호출은 a 가 valueChanged()신호를 발생하게 하고 b를 자기의 setValue()처리부에 받아들인다. 즉 b.setValue(79)가 호출된다. b는 그다음 같은 valueChanged()신호를 발생하면 b의 valuecganged()신호에 련결된 처리부가 없기때문에 아무것도 발생되지 않는다.(신호는 없어진다.)

setValue()함수는 값을 설정하며 v!=val일 때에만 신호가 발생된다. 이것은 순환련결인 경우 무한순환을 막는다. (례들어 b.valueChanged()가 a.setValue()에 련결되였을 때)

신호는 자기가 만드는 매개 련결에 대하여 발생되므로 련결이 중복되면 두 신호가 발생된다. QObject::disconnect()에 의하여 항상 련결을 없앨수 있다.

이 실례는 누군가가 처음에 객체들사이에 련결을 설정하였으면 그것들이 서로 모르고 작업할수 있다는것을 설명한다.

콤파일러가 표준 C++로 제시할수 있도록 앞처리프로그람은 signal과 slot, emit예약어를 변화시키거나 삭제한다.

신호와 처리부를 포함하고있는 클라스선언부에서 moc를 실행한다. 이것은 응용프로그람용으로 다른 목적파일들과 함께 콤파일되거나 련결되여야 하는 C++원천파일을 생

성한다. qmake를 사용한다면 moc를 자동호출하기 위한 makefile규칙들이 자기의 makefile에 추가된다.

2. 신호

신호는 객체의 의뢰기 혹은 소유자에 관심이 있도록 객체의 초기상태가 변화되였을 때 객체에 의해 발생된다. 신호를 정의하는 클라스와 그 파생클라스들만이 신호를 발생할수 있다.

실례로 목록칸은 clicked()와 currentChanged()신호를 둘다 발생한다. 거의 모든 객체들은 아마도 사용자가 항목을 마우스로 누르거나 방향건으로 이동할 때마다 현재 목록항목을 주는 currentChanged()에만 관심을 가진다. 그러나 일부 객체들은 어느 항목이 선택되였는가만 알려고 한다. 신호가 두개의 서로 다른 객체들과 관련되여있으면 두 객체의 처리부들에 신호를 련결한다

신호가 발생되면 그에 련결된 처리부는 보통 함수호출 때와 같이 즉시 실행된다. 신호-처리부기구는 어떤 GUI사건순환고리와도 완전독립이다. 모든 처리부들이 되돌아갈 때 emit도 되돌아간다.

여러개의 처리부들이 한 신호에 련결되면 신호가 발생될 때 처리부들은 임의의 순서로 하나씩 실행된다.

신호들은 자동적으로 moc에 의해 생성되므로 .cpp파일에서 실현하지 말아야 한다. 그것들은 절대로 돌림값형을 가질수 없다. 즉 void를 사용한다.

인수에 대한 알아두기: 경험은 신호와 처리부가 특별한 형을 사용하지 않으면 그것들을 재리용할수 있다는것을 보여준다.QScrollBar::valueChanged()가 가정적인 QRangeControl::Range와 같은 특수형을 사용하면 이것은 QRangeControl을 위해 특별히 설계된 처리부들에만 련결될수 있다.

3. 처리부

처리부는 거기에 련결된 신호가 발생될 때 호출된다. 처리부는 보통 C++함수들처럼 호출할수 있으며 그 유일한 특징은 신호를 처리부에 련결할수 있다는것이다. 처리부의 인수는 기정값을 가질수 없으며 신호와 같이 처리부인수로서 자체의 사용자정의형을 사용하는것은 드물다.

처리부들은 좀 추가적인 특성을 가지는 보통 성원함수들이므로 일반성원함수들과 같은 호출권을 가진다. 처리부의 호출권(access right)은 거기에 무엇을 련결할수 있는가 결정한다.

공개처리부는 신호와 련결할수 있는 처리부를 포함한다. 이것은 부분품프로그람작성에 매우 유리하며 서로 모르는 객체들을 창조하고 정보가 정확히 넘어가도록 신호와 처리부들을 련결하고 투입하고 실행한다.

보호처리부는 이 클라스와 파생클라스들이 신호를 련결할수 있는 처리부를 포함한다. 이것은 외부세계에 대한 대면부보다도 클라스의 실현부를 이루는 처리부들에 해당한 것이다.

비공개처리부는 오직 클라스자체가 신호들을 련결할수 있는 처리부들을 포함한다. 이것은 매우 단단히 련결된 클라스들을 위한것이며 여기서는 파생클라스들이 옳게 련결되였는지 알수 없다.

또한 처리부도 가상으로 정의할수 있으며 이것은 실천에서 아주 유리하다.

신호-처리부기구는 효과적이지만 역호출처럼 빠르지 못하다. 신호와 처리부는 그것들이 제공하는 유연성의 증가로 인하여 현저하게 느리지만 실제의 응용프로그람들에서 그 차이는 중요하지 않다. 일반적으로 일부 처리부들에 련결되는 신호의 발생은 비가상함수호출을 가지고 직접 수신자들을 호출하는것보다 10배정도 더 느리다. 이것은 련결

객체를 찾기 위하여, 모든 련결을 안전하게 순환하기 위하여 (즉 다음의 수신자들이 발생시에 해체되지 않았는가를 검사하면서) 그리고 일반적인 방식으로 파라메터들을 조립하기 위하여 요구되는 추가비용이다. 10개의 비가상함수호출이 많아보이지만 new 혹은 delete연산보다 비용이 훨씬 적다. 배경에서 new 혹은 delete를 요구하는 string, vector 혹은 list조작을 수행할 때 신호와 처리부들의 추가비용은 완전함수호출비의 극히 작은 비률에 대응된다. 이것은 처리부에서 체계호출을 수행할 때나 10개이상의 함수들을 간접호출할 때나 마찬가지이다. i586-500에서 매초마다 한개의 수신자에 련결된 약 2,000,000개의 신호를 발생할수 있으며 매초마다 두개의 수신자에 련결된 약 1,200,000개 신호를 발생할수 있다.

4. 메타객체정보

메타객체콤파일러(moc)는 C++파일에서 클라스선언을 해석하고 메타객체를 초기화하는 C++코드를 생성한다. 메타객체는 이 함수들의 지적자뿐아니라 모든 신호와 처리부 성원들의 이름들을 포함한다.

메타객체는 객체의 클라스이름과 같은 추가정보를 포함한다. 례들어 객체가 특정한 클라스를 계승하는가 검사할수도 있다.

```
if ( widget->inherits("QButton") ) {
  // yes, it is a push button, radio button etc.
}
```

5. 리상적인 실례

여기에 간단한 실례가 있다.

```
// qlcdnumber.h로부터 발취
  #include "qframe.h"
  #include "qbitarray.h"

  class QLCDNumber : public QFrame
```

QLCDNumber는 QObject를 계승하는데 여기에는 QFrame과 QWidget, #include의 관련한 선언들을 거쳐 신호-처리부지식을 모두 포함한다.

```
  {
    Q_OBJECT
```

Q_QBJECT는 moc에 의해 실현되는 여러 성원함수들을 선언하기 위하여 앞처리기에 의해 전개된다.《virtual function QButton::className not defined》라는 콤파일러오유를 얻으면 moc실행을 잊었거나 link지령에서 moc출력을 포함하지 않았다는것을 의미한다.

```
  public:
    QLCDNumber( QWidget *parent=0, const char *name=0 );
    QLCDNumber( uint numDigits, QWidget *parent=0, const char *name=0 );
```

moc와 명백히 관련되지 않지만 QWidget를 계승하면 거의 확정적으로 구성자들에서 *parent*와 *name*인수를 가지려고 하며 부모의 구성자에 그것들을 넘기려고 한다.

일부 해체자와 성원함수들은 여기서 생략한다. moc는 성원함수들을 무시한다.

```
  signals:
    void  overflow();
```

QLCDNumber는 불가능한 값을 현시할데 대한 요구를 받았을 때 신호를 발생한다.

자리넘침에 주의를 돌리지 않거나 자리넘침이 발생할수 없다는것을 알게 되면 자리

넘침신호를 무시할수 있다. 즉 어떤 처리부에도 그것을 련결하지 않는다.

다른 한편 수의 자리넘침이 있을 때 2개의 서로 다른 오유함수를 호출하려고 한다면 2개 다른 처리부에 그 신호를 간단히 련결한다. Qt는 임의의 순서로 둘다 호출한다.

```
    public slots:
        void  display( int num );
        void  display( double num );
        void  display( const char *str );
        void  setHexMode();
        void  setDecMode();
        void  setOctMode();
        void  setBinMode();
        void  smallDecimalPoint( bool );
    };
```

처리부는 수신함수이며 다른 창문부품들에서의 상태변화에 대한 정보를 얻는데 사용된다. QLCDNumber는 우에서 보여주는 코드와 같이 처리부를 사용하여 현시할 수값을 설정한다. display()는 프로그람의 클라스대면부이므로 처리부는 공개이다.

여러개의 실례프로그람은 display()처리부에 QScrollBar의 newValue()신호를 련결하므로 LCDNumber수값은 련속적으로 흘림띠의 값을 표시한다.

display()는 재정의되면 Qt는 처리부에 신호를 련결할 때 적당한 판을 선택한다. 역호출과 함께 5가지 서로 다른 이름들을 찾아야 하며 자체로 형들의 궤적을 보관해야한다

일부 관계없는 성원함수들은 이 실례에서 설명하지 않는다.

제4절. 메타객체체계

Qt의 메타객체체계(Meta Object System)는 객체내통신, 실행시형정보, 동적속성체계를 위한 신호-처리부기구를 제공한다.

메타객체체계는 다음의 3가지에 기초한다.

① QObject클라스,

② 클라스선언의 비공개부안에 있는 Q_OBJECT마크로,

③ 메타객체콤파일러(Meta Object Compiler) moc.

*moc*는 C++원천파일을 읽어들인다. Q_OBJECT마크로를 포함하는 하나이상의 클라스선언을 찾으면 Q_OBJECT마크로를 포함하는 클라스들을 위한 메타객체코드를 포함하는 또 다른C++원천파일을 생성한다. 이렇게 생성된 원천파일은 클라스의 원천파일에 포함되거나 클라스의 실현시에 콤파일되고 련결된다.

객체들사이의 통신을 위한 신호-처리부기구를 제공하는것(기본리유는 체계를 받아들이기 위하여)과 함께 메타객체코드는 QObject에 추가적인 기능을 제공한다.

- C++콤파일러를 통한 본래의 실행시형정보(RTTI)지원을 요구하지 않고 실행시에 문자렬로서 클라스이름을 돌려주는 className()함수.
- 객체가 QObject계승나무안의 지정된 클라스를 계승하는 클라스의 실례인가 아닌가를 돌려주는 inherits()함수 .
- 국제화에 쓰일 때 문자렬번역용의 tr()와 trUtf8()함수들.
- 이름에 의하여 객체속성들을 동적으로 설정하고 얻기 위한 setProperty()와 property()함수들.
- 그 클라스와 련관된 메타객체를 돌려주는 metaObject()함수.

Q_OBJECT마크로없이 그리고 메타객체코드없이 기초클라스로서 QObject를 사용할수 있고 한편 Q_OBJECT마크로를 사용하지 않으면 여기서 서술한 신호와 처리부나 기타 기능들을 사용할수 없다. 메타객체체계의 견지로부터 메타코드가 없는 QObject파생클라스는 메타객체코드를 가지는 그의 가장 가까운 선조와 등가하다. 이것은 실례로 className()이 자기 클라스의 실제이름을 돌려주지 않지만 선조의 클라스이름을 돌려준다는것을 의미한다. QObject의 모든 파생클라스들이 신호, 처리부, 속성들을 실제로 사용하는가 안하는가에 관계없이 Q_OBJECT마크로를 사용할것을 강하게 권고한다.

제5절. 속성

Qt는 콤파일러제작자들에 의하여 공급되는것과 비슷한 복잡한 속성(property)체계를 제공한다. 하지만 콤파일러 및가동환경에 의존하지 않는 서고처럼 Qt는 __property 혹은 [property]와 같은 비표준콤파일러기능에 기초할수 없다. 해결대책은 지원하는 모든 가동환경에 대하여 표준 C++콤파일러와 작업하는것이다. 그것은 신호와 처리부를 통한 객체통신도 제공하는 메타객체체계에 기초하고있다.

클라스선언안의 Q_PROPERTY마크로는 속성을 선언한다. 속성들은 QObject를 계승하는 클라스들에서만 선언될수 있다. 둘째 마크로 Q_OVERRIDE는 파생클라스에서 계승된 속성의 일부를 재정의하는데 쓰일수 있다.

속성은 외부세계에서 자료성원과 비슷하다. 그러나 속성들은 보통자료성원들과 구별되는 여러가지 기능을 가지고 있다.

- 읽기함수. 이것은 항상 존재한다.
- 쓰기함수. 이것은 선택적이다. QWidget::isDesktop()와 같은 읽기전용속성들은 쓰기함수를 가지지 않는다.
- 영속성을 가리키는 《기억된》특성. 대부분의 속성들은 보관되지만 일부 가상속성들은 그렇지 않다. 실례로 QWidget::minimumWidth()는 보관되지 않는다. 왜냐하면 그것이 QWidget::minimumSize()의 보기이고 자체의 자료를 가지지 않기때문이다.
- 상황에 고유한 기정값으로 속성을 설정하는 재설정함수이다.
- 속성을 GUI구축도구(실례로 Qt Designer)에서 유효로 하는것이 의의있는가를 가리키는《설계가능한》 특성. 대부분의 속성들에서 이것은 의미있지만 모두가 그런것은 아니다. 실례로 QButton::isDown()을 들수 있다. 사용자는 단추들을 누를수 있으며 프로그람작성자는 프로그람이 자체의 단추를 누르게 할수 있으나 GUI설계도구는 단추를 누를수 없다.

읽기, 쓰기 및 재설정함수들은 속성이 정의되는 클라스의 공개성원함수들이여야 한다. 속성들은 사용중에 있는 클라스에 대하여 아무것도 모르고도 QObject안의 일반함수들을 통하여 읽고 씌여질수 있다. 이 두개의 함수호출은 등가하다:

```
// QButton *b과 QObject *o는 같은 단추를 가리킨다
b->setDown( TRUE );
o->setProperty( "down", TRUE );
```

하지만 첫번째가 더 빠르고 콤파일시에 훨씬 더 좋은 진단기능을 제공한다. 실천에서는 첫번째가 더 좋다. 그러나 QMetaObject를 통하여 QObject에 대한 사용가능한 모든 속성들의 목록을 얻을수 있으므로 QObject::setProperty()는 콤파일시에 사용할수 없는 클라스들에 대한 조종을 작성자에게 넘긴다.

QObject::setProperty()에는 물론 대응하는 QObject::property()함수가 없다. QMetaObject::propertyNames()는 사용가능한 모든 속성들의 이름을 돌려준다. QMetaObject::property()는 이름있는 속성용의 속성자료 즉 QMetaProperty객체를

돌려준다.
여기에 가장 중요한 속성함수들을 보여주는 간단한 실례가 있다.

```
class MyClass : public QObject
{
  Q_OBJECT
public:
  MyClass( QObject * parent=0, const char * name=0 );
  ~MyClass();

  enum Priority { High, Low, VeryHigh, VeryLow };
  void setPriority( Priority );
  Priority priority() const;
};
```

그 클라스는 아직 메타객체체계에 알려지지 않은 속성 "priority"를 가진다. 알려진 속성을 만들려면 그것을 Q_PROPERTY마크로에 의해 선언해야 한다. 문법은 다음과 같다.

```
Q_PROPERTY( type name READ getFunction [WRITE setFunction]
    [RESET resetFunction] [DESIGNABLE bool]
    [SCRIPTABLE bool] [STORED bool] )
```

선언을 유효화하기 위하여 get함수는 const이고 형 그자체, 그것의 지적자 혹은 그에 대한 참고를 돌려주어야 한다. 선택적인 write함수는 void를 돌려주고 한개 인수형 즉 그자체, 지적자 혹은 그것에로의 const참고를 정확히 가져야 한다. 메타객체콤파일러는 이것을 수행한다.
속성의 형은 클라스자체에서 선언된 QVariant형이나 렬거형일수 있다. MyClass가 속성에 대하여 렬거형 Priority를 사용하므로 이 형은 속성체계에 등록되여야 한다.
여기에 두가지 례외가 있다. 즉 속성의 형은 QValueList<QVariant> 혹은 QMap<QString,QVariant>일수 있다. 이러한 경우에 형은 QValueList 혹은 QMap(즉 형판인수없이)로서 지정되여야 한다.
다음과 같이 이름에 의하여 값을 설정할수 있다.

```
obj->setProperty( "priority", "VeryHigh" );
```

QValueList와 QMap속성들의 경우에 넘기는 값은 완전한 list나 map값을 가지는 QVariant이다.
렬거형들은 Q_ENUMS마크로에 의해 등록된다. 여기에 속성과 관련한 선언들을 포함하는 최종적인 클라스선언이 있다.

```
class MyClass : public QObject
{
  Q_OBJECT
  Q_PROPERTY( Priority priority READ priority WRITE setPriority )
  Q_ENUMS( Priority )
public:
  MyClass( QObject * parent=0, const char * name=0 );
  ~MyClass();

  enum Priority { High, Low, VeryHigh, VeryLow };
  void setPriority( Priority );
  Priority priority() const;
```

};

다른 비슷한 마크로는 Q_SETS이다. Q_SETS는 Q_ENUMS처럼 렬거형을 등록하지만 또한 그것을 "set"로서 표식한다. 즉 렬거값들은 모두 론리합할수 있다. I/O클라스는 렬거값 "Read"와 "Write"를 가질수 있으며 "Read|Write"를 받아들일수 있다. 그러한 enum은 Q_ENUMS보다도 Q_SETS에 의해 가장 잘 조종된다.

Q_PROPERTY부분의 나머지 예약어들은 RESET, DESIGNABLE, SCRIPTABLE 및 STORED이다.

RESET는 속성을 기정상태(초기화후 변경할수 있다.)로 설정하는 함수의 이름을 짓는다. 그 함수는 void를 돌려주고 인수를 가지지 말아야 한다.

DESIGNABLE은 속성이 GUI설계도구에 의한 수정에 적합하다는것을 선언한다. 기정값은 써넣을수 있는 속성들인 경우에 TRUE이고 그렇지 않으면 FALSE이다. TRUE나 FALSE대신에 boolean성원함수를 지정할수 있다.

SCRIPTABLE은 이 속성이 스크립팅엔진에 의해 접근하는데 적합하다는것을 선언한다. 기정값은 TRUE이다. TRUE나 FALSE대신에 boolean성원함수를 지정할수 있다.

STORED는 객체의 상태를 보관할 때 속성의 값을 보관해야 하는가를 선언한다. 보관된것은 써넣기가능속성들에 대하여서만 의미를 가진다. 기정값은 TRUE이다. 기술적으로 불필요한 속성들(QRect가 속성이라면 QPoint pos와 같다.)은 이것을 FALSE로 정의한다.

추가적인 마크로 "Q_CLASSINFO"가 속성체계에 련결된다. 이것은 클라스의 메타객체에 추가적인 이름-값 쌍들을 련결하는데 쓰일수 있다. 실례로

Q_CLASSINFO("Version", "3.0.0")

다른 메타자료와 같이 클라스정보는 메타객체를 통하여 실행시에 호출할수 있다(QMetaObject::classInfo() 참고).

- Q_OVERRIDE

QObject파생클라스를 계승할 때 그 클라스의 속성들의 일부를 무시할수 있다.

실례로 QWidget에서는 다음과 같이 정의된 autoMask속성을 가진다.

Q_PROPERTY(bool autoMask READ autoMask WRITE setAutoMask DESIGNABLE false SCRIPTABLE false)

그러나 자동마스크속성을 일부 QWidget파생클라스들에서 설계할수 있게 만들 필요가 있다. 마찬가지로 일부 클라스들은 이 속성을 서술(실례로 QSA용으로)하도록 할 필요가 있다. 이것은 파생클라스에서 그 속성의 이러한 특성들을 재정의하여 달성된다. 실례로 QCheckBox에서는 다음의 코드를 사용하여 이것을 달성하다.

Q_OVERRIDE(bool autoMask DESIGNABLE true SCRIPTABLE true)

또 하나의 실례는 QToolButton이다. 기정적으로 QToolButton은 QButton으로부터 계승되므로 읽기전용 "toggleButton"속성을 가진다.

Q_PROPERTY(bool toggleButton READ isToggleButton)

그러나 도구단추를 절환할수 있게 하려고 하므로 QToolButton에 WRITE함수를 써서 다음의 속성을 재정의하는데 사용하여 그것을 호출할수 있게 한다.

Q_OVERRIDE(bool toggleButton WRITE setToggleButton)

결과는 도구단추에 대하여 읽고쓰기 boolean속성의 toggleButton이다.

제6절. Qt는 왜 신호와 처리부에 형판을 사용하지 않는가?

간단한 대답은 Qt를 설계할 때 여러가지 콤파일러들의 불충분성으로 인하여 여러가동환경에서의 형판기구를 완전히 개발할수 없었기때문이다. 오늘까지도 널리 쓰이는 많은 C++콤파일러들은 고급한 형판들과 관련한 문제를 가지고있다. 실례로 부분적인 형판실례작성에 안전하게 의거할수 없다. 이리하여 Qt에서 형판의 사용은상당히 신중해야한다. Qt가 여러 가동환경 도구일식이고 Linux/g++가동환경에서의 진보가 결코 다른 곳의 상황을 개선하지 못한다는것을 명심하여야 한다.

우연히 연약한 형판실현을 갖춘 콤파일러들이 개선된다. 그러나 모든 사용자들이 우수한 형판지원을 갖춘 완전히 표준이고 친절한 현대 C++ 콤파일러를 호출하였다고 하여도 메타객체콤파일러에 의해 사용된 문자렬에 기초한 수법을 버릴수는 없다. 여기에 5가지 리유가 있다.

1. 문법문제

알고리듬을 표시하는데 쓰이는 문법은 코드의 읽기편리성과 관리능력에 크게 영향을 주지 않는다. Qt의 신호와 처리부에 사용되는 문법은 실천에서 아주 성공적이라는것이 증명되였다. 문법은 직관적이고 사용법이 단순하고 읽기 쉽다. Qt를 배우는 사람들은문법이 고도로 추상적이고 일반적인것임에도 불구하고 신호와 처리부개념을 리해하고 사용해야 한다는것을 알고있다. 더우기 클라스정의에서 신호의 선언은신호가 보호 C++성원함수라는 의미에서 보호된다는것을 담보한다. 이것은 프로그람작성자들이 거의 처음부터 설계견본에 대하여 생각하지 않고도 정확한 설계를 얻을수 있도록 방조한다.

2. 사전콤파일러가 좋다

Qt의 moc(메타객체콤파일러)는 콤파일된 언어의 자원으로 넘기는 명백한 방법을 제공한다. 임의의 표준 C++콤파일러로 콤파일할수 있는 추가적인 C++코드를 생성하여 그렇게 할수 있다. moc는 C++원천파일들을 읽어들인다. Q_OBJECT마크로를 포함하는 하나이상의 클라스선언을 발견하면 그 클라스들을 위한 메타객체코드를 포함하는 다른 C++원천파일을 생성한다. moc가 생성한 C++원천파일은 클라스의 실현과 함께 콤파일되고 련결된다. (혹은 클라스의 원천파일에 #include될수 있다.) 일반적으로 moc는 수동적으로 호출되지 않고 구축체계에 의해 자동적으로 호출되므로 프로그람작성자의 추가적인 작업을 요구하지 않는다.

다른 사전콤파일러들 실례로 rpc와 idl은 프로그람이나 객체들이 프로쎄스나 콤퓨터에서 통신할수 있게 한다. 사전콤파일러들에서 한가지 문제는 선명치 않은 코드를 생성하는 대화칸나 위자드를 갖추고있는 콤파일러들, 저작권이 있는 언어들 혹은 도형방식프로그람작성도구들을 포기하는것이다. 저작권이 있는 C++ 콤파일러나 특정한 통합개발환경에 손님들이 포로되게 하는것이 아니라 그들이 자기가 좋아하는 어떤 도구든지 사용할수 있게 한다.

3. 유연성이 제일이다

C++는 표준화되고 강력하고 정성들여 만든 일반목적언어이다. 이것은 쏘프트웨어프로젝트, 전체 조작체계들에 대해 동작하는 각종 응용프로그람의 전개, 자료기지봉사기와 탁상응용프로그람들에서 일반적인 고말단 도형방식응용프로그람과 같은 넓은 범위에서 개발되는 유일한 언어이다. C++성공의 열쇠들중의 하나는 ANSI-C 호환성을 계속 유지하면서 최대성능과 최소기억기소비에 초점을 둔 확장가능한(scalable) 언어설계이다.

이러한 모든 우점들에 대조되는 결함이 있다. C++에서 정적객체모형은 부분품에 기초한 도형방식사용자대면부프로그람을 작성할 때 Objective C의 동적통보전송수법에 대한 명백한 결함이다. 고말단자료기지봉사기 혹은 조작체계가 결코 GUI첨단의 옳은 설계선택은 아니다. moc를 사용하여 이 결함을 우점으로 전환하고 안전하고 효과적인 도형방식사용자대면부 프로그람작성에 필요한 유연성을 추가하였다.

우리의 수법은 형판에 의해 수행할수 있는 일에 아주 효과있다. 실례로 객체속성들을 가질수 있다. 그리고 신호와 처리부를 재정의하여 재정의가 주개념으로 되여있는 언어로 프로그람작성할 때 자연스러운것으로 할수 있다. 신호는 클라스실례에 0byte를 추가한다. 이것은 2진호환성을 파괴함이 없이 새 신호들을 추가할수 있다는것을 의미한다. 형판에 의해 수행한것처럼 과도한 즉시화(inlining)에 의거하지 않으므로 코드크기를 작게 할수 있다. 새로운 련결의 추가는 복잡한 형판함수가 아니라 단순한 함수호출로 전개된다.

다른 리득은 실행시에 객체의 신호와 처리부들을 탐구할수 있는것이다. 련결하고있는 객체들의 정확한 형을 모르고도 형안전하고 이름에 의한 호출을 리용하여 련결을 확립할수 있다. 이것은 형판에 기초한 해결로서는 불가능하다. 이러한 종류의 실행시 자기관찰(introspection)은 새로운 가능성을 열어놓는다. 실례로 Qt Designer의 XML ui 파일들로부터 생성되고 련결되는 GUI들.

4. 성능호출이 모든것은 아니다

Qt의 신호와 처리부실현은 형판에 기초한 해결처럼 빠르지 못하다. 신호발생비용은 거의 일반형판실현에 의한 4개의 보통함수호출의 비용이지만 Qt는 10개의 함수호출에 대하여 상당한 작업을 요구한다. 이것은 Qt기구가 일반조립기(marshaller), 자기관찰과 극도의 스크립트가능성을 포함하므로 놀랄만한것이 못된다. 이것은 과도한 즉시화와 코드팽창에 의거하지 않고 실행시안전성을 제공한다. Qt의 반복자는 안전하지만 고속형판에 기초한 체계의 반복자는 그렇지 못하다. 여러개의 수신자들에 신호를 발생하는 처리과정에 수신자들은 자기 프로그람을 중단함이 없이 안전하게 삭제될수 있다. 이러한 안전성이 없이는 자기의 응용프로그람이 해방된 기억기의 읽기 혹은 써넣기오유를 오유수정하기 어려운것으로 하여 우연히 중단될것이다.

그럼에도 불구하고 형판에 기초한 해결이 신호와 처리부를 리용하는 응용프로그람의 성능을 개선할수 없는가? Qt는 신호를 통한 처리부호출비에 적은 비용이 추가되고 그 호출비는 처리부의 총비용에서 작은 몫을 차지한다. 일반적으로 Qt의 신호-처리부체계에 대한 성능시험은 빈 처리부를 가지고 수행된다. 자기의 처리부에 사용할수 있는 어떤 일 실례로 좀 단순한 문자렬조작을 할 때 호출간접비는 무시된다. Qt체계는 연산자 new 혹은 delete를 요구하는 조작(실례로 문자렬조작 혹은 형판용기로부터 무엇인가 삽입 및 삭제)이 신호발생보다 훨씬 더 많은 비용이 드므로 최적화한다.

례외: 성능이 림계상태에 이르는 과제의 엄격한 내부순환안에 신호와 처리부가 있고 그 련결이 난관이라는것을 확인하였다면 신호와 처리부보다도 표준 청취기(listener) 대면부를 사용한다. 이러한 일이 생기는 경우에 1:1련결만 요구한다. 실례로 망으로부터 자료를 내리적재하는 객체가 있다면 요구하는 자료가 도착하였다는것을 가리키는데 신호를 사용하는것은 완전히 좋은 설계이다. 그러나 소비자에게 매번 1byte씩 보내야 한다면 신호와 처리부보다도 표준 청취기대면부를 사용한다.

5. 제한이 없다

신호와 처리부용 moc가 있으므로 형판로 수행할수 없는것에 다른 유용한것들을 추가할수 있다. 그중에는 생성된 tr()함수를 거치는 유효범위있는 번역과 자기관찰을 갖추고있는 고급한 속성체계 및 확장된 실행시형정보가 있다. 속성체계는 큰 우점을 가지고 있다. 즉 강력하고 자기관찰적인 속성체계가 없이 Qt Designer와 같은 강력하고 일반적인 사용자대면부설계도구를 작성하는것은 훨씬 더 힘들다.

moc앞처리기를 갖춘 C++는 본질적으로 우리에게 Objective-C 혹은 Java실행시환경의 유연성을 주는 한편 C++의 독특한 성능과 확장가능성의 우점을 유지한다. 이것이 오늘 Qt를 유연하게 하고 편리한 도구로 만들고있다.

제4장. Qt의 리용방법

제1절. 설치

Qt의 설치절차는 가동환경에 따라 다르다.

1. Qt/X11의 설치

Qt를 설치하려는 등록부의 사용허가에 따라서 root로 로그인할 필요가 있다.

① 압축파일을 푼다.(이미 풀지 않았으면)

```
cd /usr/local
gunzip qt-x11-version.tar.gz  # uncompress the archive
tar xf qt-x11-version.tar     # unpack it
```

이것은 기본압축파일의 파일들을 포함할 등록부 /usr/local/qt-*version*을 창조한다.

이름을 qt-*version*으로부터 qt로 바꾼다. (혹은 기호련결을 만든다.)

```
mv qt-version qt
```

나머지 파일은 Qt가 /usr/local/qt에 설치된다고 가정한다.

② 파일 .profile (혹은 자기의 쉘에 따라서 .login)안의 일부 환경변수들을 자기의 home등록부로 설정한다. 그 파일이 거기에 없으면 창조한다.

QTDIR -- Qt를 구축하고있는 등록부,

PATH -- moc프로그람과 다른 Qt도구들을 배치하는 경로,

MANPATH -- Qt기본페지들에 접근하는 경로,

LD_LIBRARY_PATH -- 공유Qt서고의 경로.

이것은 다음과 같이 수행된다.

.profile에 (쉘이 bash, ksh, zsh 혹은 sh이면) 다음과 같은 행들을 추가한다.

```
QTDIR=/usr/local/qt
PATH=$QTDIR/bin:$PATH
MANPATH=$QTDIR/man:$MANPATH
LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
export QTDIR PATH MANPATH LD_LIBRARY_PATH
```

.login에 (쉘이 csh나 tcsh인 경우에) 다음과 같은 행들을 추가한다.

```
setenv QTDIR /usr/local/qt
setenv PATH $QTDIR/bin:$PATH
setenv MANPATH $QTDIR/man:$MANPATH
setenv LD_LIBRARY_PATH $QTDIR/lib:$LD_LIBRARY_PATH
```

그다음 다시 로그인하거나 혹은 계속하기전에 프로파일의 원천을 재구성하여 $QTDIR를 설정한다.

LD_LIBRARY_PATH대신에 AIX에서는 LIBPATH를, HP-UX에서는 SHLIB_PATH를 설정한다.

SGI MIPSpro o32와 Sun WorkShop 5.0목표들은 Qt 3.3에서와 같이 더는 지원되지 않는다.

③ $HOME/.qt-license로서 자기 허가파일을 설치한다. 무료판과 평가판에서는 허가파일이 필요없다.

④ Qt서고를 콤파일하고 실례프로그람, 련습, 도구(실례로 Qt Designer)를 다음과 같이 입력하여 구축한다.

./configure

이것은 자기 콤퓨터로부터 Qt서고환경을 구성한다. GIF유지는 기정으로 차단된다. ./configure -help를 실행하여 환경선택목록을 얻는다. 지원하고있는 가동환경목록용의 PLATFORMS을 읽는다.

서고를 창조하고 모든 실례와 련습을 콤파일하기 위하여 다음과 같이 입력한다.

make

⑤ 아주 적은 경우에 공유서고를 사용하고있으면 이 시점에서 /sbin/ldconfig이나 그와 류사한것을 실행할 필요가 있다.

실례프로그람들을 실행하는데서 문제가 생기면 실례로다음과 같은 통보가 생기면

can't load library 'libqt.so.2'

환경파일에 qt서고에로의 참고를 넣고 자기 체계에서 루트로서 /sbin/ldconfig를 실행할 필요가 있다. 그리고 우의 ②에서 설명한것처럼 LD_LIBRARY_PATH를 설정한다.

⑥ 직결HTML문서는 /usr/local/qt/doc/html/에 설치된다. 기본페지는 /usr/local/qt/doc/html/index.html이다. man페지들은 /usr/local/qt/doc/man/에 설치된다. 또한 문서는 Qt Assistant에 의해 호출할수도 있다.

이제는 Qt가 설치된다.

2. Qt/Windows의 설치

Qt/Windows배포물은 기본설치프로그람을 갖춘 자체발취압축파일로서 배포된다. 그러므로 설치위자드를 실행하면 설치된다.

3. Qt/Mac의 설치

Qt를 설치하는 등록부의 사용허가에 따라서 root로 로그인할 필요가 있다.

① 압축파일을 푼다.(이미 하지 않았으면)

cd /Developer

gnutar xzf qt-mac-version.tar.gz

이것은 기본압축파일로부터 파일들을 포함할 등록부 /Developer/qt-*version* 를 창조한다.

이름을 qt-*version*으로부터 qt로 바꾼다. (혹은 기호련결을 만든다.)

mv qt-version qt

나머지 파일은 Qt가 /Developer/qt에 설치된다고 가정한다.

② 파일 .profile (혹은 자기의 쉘에 따라서 .login)안의 일부 환경변수를 자기의 home등록부로 설정한다. 파일이 거기에 없으면 창조한다.

QTDIR -- Qt를 구축하고있는 등록부

PATH -- moc프로그람과 그밖에 Qt도구를 찾는 경로

MANPATH -- Qt man페지호출경로

LD_LIBRARY_PATH - 공유Qt서고용 경로

이것은 다음과 같이 수행된다.

.profile에서 (자기의 쉘이 bash, ksh, zsh 혹은 sh이면) 다음의 행들을 추가한다.

```
QTDIR=/Developer/qt
PATH=$QTDIR/bin:$PATH
MANPATH=$QTDIR/man:$MANPATH
DYLD_LIBRARY_PATH=$QTDIR/lib:$DYLD_LIBRARY_PATH
export QTDIR PATH MANPATH DYLD_LIBRARY_PATH
```

.login에서 (자기 쉘이 csh 혹은 tcsh인 경우에) 다음 행들을 추가한다.

```
setenv QTDIR /Developer/qt
```

```
setenv PATH $QTDIR/bin:$PATH
setenv MANPATH $QTDIR/man:$MANPATH
setenv DYLD_LIBRARY_PATH $QTDIR/lib:$DYLD_LIBRARY_PATH
```

그 다음에 다시 로그인하거나 계속하기전에 프로파일원천을 다시 만들수 있다.

③ Qt의 상업판을 가지고있다면 자기의 사용허가파일을 $HOME/.qt-license로서 설치한다.

④ Qt서고를 콤파일하고 실례프로그람들과 련습, 도구들(실례로 Qt Designer)을 다음과 같이 구축한다.

입력:

```
cd $QTDIR
./configure
```

이것은 자기 콤퓨터용의 Qt서고환경을 구성한다. GIF기능은 기정으로 설정되지 않는다. ./configure –help를 실행하여 환경선택목록을 얻는다.

서고를 창조하고 모든 실례와 련습을 콤파일하려면

```
make
```

⑤ 일단 Qt를 구축하였다면 사용할 준비가 된다. Qt가 –static선택으로 환경구성되지 않았다면 Finder로부터 Qt응용프로그람들을 호출할수 있게 하기 위하여 탐색등록부에 관련서고들을 배치해야 한다. 그 기호련결을 만들것을 권고한다. (–thread를 가지고 Qt환경을 구성하였다면 아래의 libqt를 libqt-mt로 변경한다.)

```
ln -sf $QTDIR/lib/libqt.3.dylib /usr/lib
ln -sf $QTDIR/lib/libqui.1.dylib /usr/lib
```

이를 위하여 체계관리자호출을 가질 필요가 있다. (그 경우에 매개 지령앞에 sudo를 놓는다. 실례로 sudo ln -s ...). sudo를 사용한다면 체계관리자의 암호를 묻는다.

체계관리자호출을 가지지 않거나 혹은 서고의 사용자설치(체계설치가 아니라)를 좋아한다면 다음과 같이 할수 있다. (–thread를 가지고 Qt환경을 구성하였다면 아래의 libqt를 libqt-mt로 변경한다.)

```
ln -sf $QTDIR/lib/libqt.3.dylib $HOME/lib
ln -sf $QTDIR/lib/libqui.1.dylib $HOME/lib
```

이제는 Qt가 설치된다. Qt의 문서는 Qt Assistant 혹은 다른 웨브열람기로 읽어들일수 있으며 내용페지는 $QTDIR/doc/html/index.html로 된다.

제2절. Qt플라그인

Qt는 사용자정의자료기지구동프로그람, 화상형식, 본문코드, 콤포넨트로 이루어져있는 양식기능들과 도구들을 창조하기 쉽게 하는 간단한 플라그인대면부를 제공한다.

플라그인작성은 적합한 플라그인기초클라스의 파생클라스를 만들고 일부 함수들을 실현하고 마크로를 추가함으로써 달성된다.

표 4-1에 5개의 플라그인기초클라스가 있다. 파생된 플라그인들은 기정으로 표준플라그인등록부에 보관된다.

표 4-1. 플라그인기초클라스들

기초클라스	기정경로
QImageFormatPlugin	pluginsbase/imageformats
QSqlDriverPlugin	pluginsbase/sqldrivers
QStylePlugin	pluginsbase/styles
QTextCodecPlugin	pluginsbase/codecs
QWidgetPlugin	pluginsbase/designer

그러면 pluginsbase등록부는 어디에 있는가? 응용프로그람을 실행할 때 Qt는 우선 응용프로그람의 실행가능한 등록부를 pluginsbase로서 취급한다. 실례로 응용프로그람이 플라그인형식을 가지고 있다면 C:\Program Files\MyApp\styles를 찾는다. (응용프로그람의 실행파일위치를 찾는 방법은 QApplication::applicationDirPath()을 참고하시오.) 또한 Qt는 qInstallPathPlugins()에 의하여 주어지는 등록부를 얻는다. Qt가 추가적인 위치를 보게 하려면 QApplication::addLibraryPath()호출을 사용하여 필요한만큼 경로를 추가해야 한다. 그리고 자체의 경로를 하나 또는 여러개 설정하려면 QApplication::setLibraryPaths()를 사용해야 한다.

플라그인으로 쓸수있게 만들려는 MyStyle라는 새 형식클라스를 가지고있다고 가정하자. 필요한 코드는 간단하다.

```
class MyStylePlugin : public QStylePlugin
{
public:
  MyStylePlugin() {}
  ~MyStylePlugin() {}

  QStringList keys() const {
    return QStringList() << "mystyle";
  }

  QStyle* create( const QString& key ) {
    if ( key == "mystyle" )
      return new MyStyle;
    return 0;
  }
};

Q_EXPORT_PLUGIN( MyStylePlugin )
```

(QStyleFactory는 대소문자를 구별하고 소문자건을 사용하며 다른 공장 실례로 QWidgetFactory는 대소문자를 구별한다.)

구성자와 해체자는 아무것도 할 필요가 없으므로 비여있다. 실현해야 할 가상함수는 두개이다. 첫번째는 플라그인에서 실현된 클라스들의 문자렬목록을 돌려주는 keys()함수이다. (우의 실례에서 하나의 클라스를 실현하였다.) 두번째는 요구되는 클라스의 객체(혹은 플라그인이 실현하지 않은 클라스의 객체를 창조할것을 요구한다면 0)를 돌려주는 함수이다. QStylePlugin에서 두번째함수는 create()이다.

QStylePlugin. 하나의 플라그인에서 임의의 개수의 플라그인파생클라스를 실현할수 있으며 이것은 파생클라스들이 모두 같은 기초클라스 실례로 QStylePlugin으로부터 파생될수 있게 한다.

자료기지구동프로그람, 화상형식, 사용자정의창문부품, 본문코드에서는 어떤 명시적인 객체창조도 요구하지 않는다. Qt는 필요할 때 그것들을 찾아서 창조한다. 코드에서 형식을 명시적으로 설정하려고 하므로 형식(style)은 례외이다. 형식을 적용하기 위해서는 다음의 코드를 사용한다.

```
QApplication::setStyle( QStyleFactory::create( "MyStyle" ) );
```

일부 플라그인클라스들은 추가적인 함수들을 실현할것을요구한다. *Qt Designer*에 플라그인을 통합하는 특별한 함수들을 실현하는 QWidgetPlugin의 완전한 실례는 《 Qt도구의 사용법 》 1장 6절에서 설명한다. QWidgetFactory클라스는

QWidgetPlugin에 추가적인 정보를 제공한다.

플라그인은 표준플라그인보조등록부에 보관되므로 Qt응용프로그람들은 자동적으로 어느 플라그인이 유효한가를 알고있다. 그러므로 응용프로그람들은 플라그인을 발견하고 적재하는 코드를 요구하지 않는다. 즉 Qt가 자동적으로 조종한다.

플라그인의 기정등록부는 QTDIR/plugins(QTDIR에 대한 모든 참고는 Qt가 설치된 경로에 귀착된다.)인데 매개 형의 플라그인은 그 형의 보조등록부 실례로 styles안에 있다. 자기 응용프로그람들이 플라그인을 사용하려고 하는데 표준플라그인경로를 사용하지 않으려면 설치과정에 플라그인에 사용하려는 경로를 결정하게 하고 실행할 때 읽어들이는 응용프로그람을 위하여 그 경로를 보관한다. 그다음 응용프로그람은 이 경로를 리용하여 QApplication::addLibraryPath()를 호출하고 자기의 플라그인들을 응용프로그람에 사용할수 있게 된다. 경로의 마지막 부분 즉 styles, widgets 등은 변경할수 없다.

응용프로그람에 플라그인을 포함하는 일반적인 방법은 그것을 응용프로그람으로 콤파일하거나 DLL(혹은 다른 가동환경에 고유한 서고형)로 콤파일하여 다른 서고처럼 그것을 사용하는것이다. 플라그인을 적재할수 있게 하는 하나의 수법은 응용프로그람안에 보조등록부 례를 들면 appdir/plugins/designer를 창조하고 그 등록부에 플라그인을 배치하는것이다.

Qt Designer에서는 QApplication::addLibraryPath("QTDIR/plugins/designer")를 호출하여 자기의 Qt Designer플라그인들을 적재하여야 한다.

1. 플라그인의 적재와 확인

플라그인들을 적재할 때 Qt 서고는 플라그인을 적재하여 사용할수 있는가를 결정하는 검사를 한다. 이것은 나란히 설치된 Qt서고의 여러개의 판본과 환경구성을 가능하게 한다.

• 높은 기본판본이나 낮은 보조판본을 가지는 Qt서고와 련결된 플라그인들은 낮은 기본판본이나 낮은 보조판본을 가지는 서고에 의하여 적재되지 않는다.

리유: Trolltech는 보조판본의 출하사이에만 새로운 특성과 API들을 추가하는 정책을 실시하고있다. 그것은 이 시험이 패취(patch)준위판번호가 아니라 오직 기본판번호와 보조판번호를 검사하기때문이다.

• 스레드를 지원하는 Qt서고에 련결한 플라그인들은 스레드를 지원하여 구축되는 서고에 의해서만 적재될수 있다.

리유: 스레드 및 비스레드화된 Qt서고들은 다른 이름을 가진다. 비스레드 Qt서고에 련결된 플라그인을 적재하는 스레드지원서고는 두 판본의 같은 서고가 동시에 기억기에 있게 한다. UNIX체계에서 이것은 비스레드화된 Qt서고를 적재하게 한다. 이러한 일이 발생하면 Qt서고에서 모든 정적객체들의 구성자들은 두번째로 호출되지만 그것들은 벌써 기억기의 객체들에 대하여 조작한다. 이것과 작업하는 방법은 없다. 이것은 스레드화된 Qt서고에 의하여 이미 정의된 정적기호들을 비스레드화된 Qt서고가 적재될 때 교체하거나 복사할수 없는 목적2진형식의 특성이다.

• 비스레드 Qt서고에 련결된 플라그인들은 스레드를 지원하지 않고 구축되는 서고에 의하여 적재될수 있을뿐이다.

리유: 우의 리유를 보시오.

• Qt 3.0.5로 시작할 때 Qt서고와 플라그인들은 구축건(build key)에 기초하여 구축된다. Qt서고의 구축건을 플라그인의 구축건과 대조하여 그것들이 일치하면 플라그인이 적재되고 일치하지 않으면 플라그인의 적재는 거부된다.

리유: 아래의 구축건의 리유를 보시오

2. 구축건

구축건은 다음과 같은 정보를 포함한다.

• 구성방식과 조작체계, 콤파일러

리유: 서로 다른 판본의 같은 콤파일러가 2진호환코드를 생성하지 못하는 경우에 콤파일러의 판본은 구축건에 주어진다.

• Qt서고의 환경구성. 환경구성은 서고안의 유효API에 영향을 주는 보이지 않는 특성들의 목록이다.

리유: 같은 판본의 Qt서고의서로 다른 2개 환경구성은 2진호환되지 않는다. 플라그인을 적재하는 Qt서고는 (보이지 않는)특성들의 목록을 사용하여 그 플라그인이 2진호환되는가를 결정한다.

알아두기: 두가지 다른 구성방식에서 플라그인이 유효특성들을 사용할수 있는 경우가 있다. 하지만 플라그인을 쓰는 개발자는 플라그인과 Qt에서 편의클라스들에서 내적으로 어떤 특성들을 사용하고있는가 알아야 한다. Qt서고는 플라그인을 적재할 때 복잡한 기능과 의존관계질문, 검증을 요구한다. 이 요구는 개발자에게 불필요한 부담을 주고 플라그인적재의 비용을 늘인다. 개발시간과 응용프로그람실행원가를 모두 줄이기 위하여 구축건들의 간단한 문자렬대조가 사용된다.

• 선택적으로 여분의 문자렬을 configure 스크립트지령행에 지정할수 있다.

리유: 응용프로그람과 함께 Qt서고의 2진파일들을 배포할 때 이것은 개발자들에게 플라그인들이 련결된 서고에 의하여서만 적재될수 있는 플라그인작성방법을 제공한다.

3. 플라그인과 스레드응용프로그람

스레드화된 Qt서고(플라그인자체가 스레드를 사용하는가 안하는가에 따라)와 함께 사용하려는 플라그인을 구축하려고 한다면 스레드환경을 사용해야 한다. 특히 스레드Qt서고와 플라그인을 련결해야 하며 그 서고와 함께 Qt Designer를 구축해야 한다. 자기 플라그인의 .pro파일은 다음 행에 포함하여야 한다.

```
CONFIG += thread
```

경고: 응용프로그람에서 보통의 Qt서고와 스레드화된 Qt서고를 혼합하지 말아야 한다. 자기 응용프로그람이 스레드화된Qt서고를 사용한다면 자기 플라그인을 보통의 Qt서고와 련결하지 말아야 한다. 보통의 Qt서고를 동적으로 적재하거나 다른 서고 실례로 보통의 Qt서고에 의존하는 플라그인을 동적으로 적재하여서는 안된다. 일부 체계에서 스레드화 및 비스레드화된 서고들이나 플라그인들의 혼합은 Qt서고에서 사용하는 정적자료를 못쓰게 한다.

제3절. QDataStream연산자들의 형식

QDataStream는 일부 Qt자료형들을 계렬화한다. 표 4-2는 QDataStream이 계렬화하고 표시할수 있는 자료형을 렬거한다.

옹근수들을 써넣을 때 Qt옹근수로 강제변환하고 읽을 때에는 같은 Qt옹근수형으로 읽어들여야 한다.

표 4-2. Qt자료형

Q_INT8	부호있는 바이트
Q_INT16	부호있는16bit옹근수
Q_INT32	부호있는32bit옹근수
Q_UINT8	부호없는 바이트
Q_UINT16	부호없는 16bit옹근수

Q_UINT32	부호없는 32 bit옹근수
float	32bit 류동소수점수. 표준IEEE-754 형식을 사용한다.
double	64bit 류동소수점수. 표준IEEE-754형식을 사용한다.
char *	0완료를 포함하는 문자렬의 크기(Q_UINT32) 0완료를 포함하는 문자렬바이트 null문자렬은 (Q_UINT32)0으로 표시된다.
QBitArray	배렬크기(Q_UINT32) 배렬비트 즉 (size+7)/8byte
QBrush	솔형식(Q_UINT8) 솔의 색(QColor) 형식이 CustomPattern이면 솔픽스매프(QPixmap)
QByteArray	배렬크기(Q_UINT32) 배렬바이트 즉 크기바이트
QCString	0완료를 포함하는 문자렬의 크기(Q_UINT32) 0완료를 포함하는 문자렬바이트 null문자렬은 (Q_UINT32)0으로 표시된다.
QColor	Q_UINT32로서 계렬화된 RGB값
QColorGroup	foreground (QBrush) button (QBrush) light (QBrush) midLight (QBrush) dark (QBrush) mid (QBrush) text (QBrush) brightText (QBrush) ButtonText (QBrush) base (QBrush) background (QBrush) shadow (QBrush) highlight (QBrush) highlightedText (QBrush)
QCursor	형태식별자(Q_INT16) 형태가 BitmapCursor이면 즉 비트매프(QPixmap)와 마스크(QPixmap), hot spot (QPoint)
QDate	율리우스날자 (Q_UINT32)
QDateTime	날자 (QDate) 시간 (QTime)
QFont	가계 (QCString) 점크기 (Q_INT16) 형식암시 (Q_UINT8) 문자모임 (Q_UINT8) 무게 (Q_UINT8) 서체비트 (Q_UINT8)
QImage	화상이 null이면 "null image"표식이 보관되고 그렇지 않으면 화상은 스트림판에 따라 PNG 혹은 BMP형식으로 보관된

	다. 형식을 조종하려고 한다면 QImageIO를 리용하여 QBuffer에로 화상이 흐르게 한다.
QMap	항목수 (Q_UINT32) 모든 항목들에 대하여 건과 값
QPalette	active (QColorGroup) disabled (QColorGroup) inactive (QColorGroup)
QPen	펜형식 (Q_UINT8) 펜폭 (Q_UINT8) 펜색 (QColor)
QPicture	그림자료의 크기(Q_UINT32) 그림자료의 생(raw)바이트 (char)
QPixmap	PNG화상으로 보관한다.
QPoint	x자리표(Q_INT32) y자리표(Q_INT32)
QPointArray	배렬크기 (Q_UINT32) 배렬의 점 (QPoint)
QRect	left (Q_INT32) top (Q_INT32) right (Q_INT32) bottom (Q_INT32)
QRegion	자료의 크기 즉 8 + 16 * (직4각형수) (Q_UINT32) QRGN_RECTS (Q_INT32) 직4각형수 (Q_UINT32) 순차적으로 놓여있는 직4각형들 (QRect)
QSize	width (Q_INT32) height (Q_INT32)
Qstring	문자렬이 null이면 0xffffffff (Q_UINT32) 그렇지 않으면 문자렬길이(Q_UINT32)와 그뒤에 오는 UTF-16의 자료
QTime	자정으로부터의 미리초 (Q_UINT32)
QValueList	목록원소수 (Q_UINT32) 순차로 놓인 모든 원소들
QVariant	자료형(Q_UINT32) 지정된 형의 자료
QWMatrix	m11(double) m12(double) m21(double) m22(double) dx(double) dy(double)

제4절. 오유수정

여기서는 Qt에 기초하는 쏘프트웨어를 오유수정하기 위한 몇가지 쓸모있는 암시를 제시한다.

1. 지령행추가선택

Qt프로그람을 실행할 때 오유수정을 방조할수 있는 몇가지 지령행추가선택을 표 4-3에 주었다.

표 4-3. 오유수정지령행추가선택

추가선택	결과
-nograb	응용프로그람은 마우스 혹은 건반을 포획하지 말아야 한다. 이 추가선택은 Linux에서 프로그람을 오유수정기gdb에서 실행하고 있을 때 기정으로 설정된다.
-dograb	암시적이거나 명시적인 -nograb를 무시한다. -nograb가 지령행의 마지막인 경우에도 -dograb가 –nograb보다 우선권이 높다.
-sync	X동기방식에서 응용프로그람을 실행한다. 동기방식은 X봉사기가 매개 X의뢰기요구를 즉시 처리하게 하며 완충기최적화를 사용하지 않는다. 이것은 프로그람의 오유수정을 쉽게 하지만 아주 느리게 한다. -sync선택은 Qt의 X11판에서만 유효하다.

2. 경고와 오유수정통보

Qt는 경고 및 오유수정본문을 쓰기 위한 3개의 대역함수를 포함한다.

- qDebug()는 시험 등을 위한 오유수정출력을 쓴다.
- qWarning()은 프로그람오유가 발생할 때 경고출력을 쓴다.
- qFatal()은 치명적오유통보를 쓰고 완료한다.

Qt에서 이 함수들의 실현은 Unix/X11에서는 stderr출력에, Windows에서는 오유수정기에 본문을 출력한다. 통보문처리함수 qInstallMsgHandler()를 설치하여 이 함수들을 물려받을수 있다.

오유수정함수 QObject::dumpObjectTree()와 QObject::dumpObjectInfo()는 흔히 응용프로그람이 이상하게 보이거나 동작할 때 쓸모있다. 객체이름을 사용할 때 더 쓸모있지만 지어는 이름없이도 사용할수 있다.

3. 오유수정마크로

머리부파일 qglobal.h는 많은 오유수정정보와 #define들을 포함한다.

2개의 중요한 마크로가 있다. 즉

- Q_ASSERT(b), 여기서 b는 론리식이고 b가 FALSE이면 경고 "ASSERT: 'b' in file file.cpp (234)"를 출력한다.
- Q_CHECK_PTR(p), 여기서 p는 지적자이다. p가 0이면 경고 "In file file.cpp, line 234: Out of memory"를 출력한다.

이 마크로들은 프로그람오유를 탐지하는데 쓸수 있다. 례를 들면

```
char *alloc( int size )
{
  Q_ASSERT( size > 0 );
  char *p = new char[size];
  Q_CHECK_PTR( p );
  return p;
```

```
}
```

기발 QT_FATAL_ASSERT를 정의하면 Q_ASSERT는 warning()대신에 fatal()을 호출하므로 실패한 확인문(assertion)은 오유통보를 출력한 후에 프로그람을 완료시킨다. QT_CHECK_STATE가 정의되지 않으면 Q_ASSERT마크로는 null식이다. 그안의 코드는 단순히 실행되지 않는다. 마찬가지로 QT_CHECK_NULL가 정의되지 않으면 Q_CHECK_PTR는 null식이다. 여기에 Q_ASSERT와 Q_CHECK_PTR를 사용하지 말아야 하는 실례가 있다.

```
char *alloc( int size )
{
  char *p;
  Q_CHECK_PTR( p = new char[size] ); // WRONG!
  return p;
}
```

*p*는 정확한 검사값이 정의되기만 하면 그 값으로 설정된다. 이 코드가 정의된 QT_CHECK_NULL기발없이 콤파일되면 Q_CHECK_PTR식안의 코드는 실행되지 않고(정확히 오유수정목적뿐이므로) *alloc*는 임의의 지적자를 돌려준다.

Qt서고는 오유가 발견될 때 경고통보를 출력하는 수백개의 내부검사를 포함한다.

Qt내에서 제대로 되는가하는 시험과 결과에 따라 생기는 경고통보는 여러가지 오유수정기발의 상태에 기초한다(표 4-4).

표 4-4. 오유수정기발

기발	의미
QT_CHECK_STATE	비모순성과 기대한 객체상태검사
QT_CHECK_RANGE	변수범위오유검사
QT_CHECK_NULL	위험한 null지적자검사
QT_CHECK_MATH	위험한 수학오유, 실례로 0에 의한 나누기를 검사
QT_NO_CHECK	모든 QT_CHECK_...기발들을 해제
QT_DEBUG	오유수정코드허용
QT_NO_DEBUG	QT_DEBUG기발 해제

기정으로 QT_DEBUG와 모든 QT_CHECK기발들은 설정되여있다. QT_DEBUG를 해제하려면 QT_NO_DEBUG를 정의한다. QT_CHECK기발들을 해제하려면 QT_NO_CHECK를 정의한다.

실례:

```
void f( char *p, int i )
{
#if defined(QT_CHECK_NULL)
  if ( p == 0 )
    qWarning( "f: Null pointer not allowed" );
#endif

#if defined(QT_CHECK_RANGE)
  if ( i < 0 )
    qWarning( "f: The index cannot be negative" );
#endif
}
```

4. 일반오유

여기서 마땅히 언급해야 할 일반오유가 하나 있다. 즉 클라스선언에 Q_OBJECT 마크로를 선언하고 moc를 실행하지만 실행파일에 moc가 생성한 목적코드를 련결하지 않으면 오유통보를 얻는다. vtbl, _vtbl, __vtbl의 결핍이나 그와 류사한 련결오유는 이러한 문제의 결과이다.

제5절. 끌어다놓기

끌어다놓기는 사용자가 응용프로그람들사이 혹은 그 안에서 정보를 전송하는데 사용할수 있는 간단한 시각적기구를 제공한다. 끌어다놓기는 오려둠판의 자르기-붙이기기구의 기능과 비슷하다. (끌어다놓기실례들에 대하여서는 qt/examples/iconview/simple_dd, qt/examples/dragdrop, qt/examples/fileiconview, 또한 QTextEdit창문부품원천코드를 참고하시오.)

1. 끌기

실례로 마우스이동사건에서 끌기를 시작하려면 본문일 때 QTextDrag, 화상일 때 QImageDrag와 같이 자기 매체에 적당한 QDragObject파생클라스의 객체를 창조한다. 그다음 drag()메쏘드를 호출한다.

실례로 창문부품으로부터 본문의 끌기를 시작하려면 다음과 같이 한다.

```
void MyWidget::startDrag()
{
  QDragObject *d = new QTextDrag( myHighlightedText(), this );
  d->dragCopy();
 // do NOT delete d.
}
```

QDragObject는 끌기후에 삭제되지 않는다. QDragObject는 아직 다른 프로쎄스와 교제하고있을수 있으므로 끌기가 명백히 끝난 후에도 존재할 필요가 있다. 우연적으로 Qt는 그 객체를 삭제한다. 끌기객체를 소유하는 창문부품이 그전에 삭제되면 종속된 놓기가 취소되고 끌기객체는 삭제된다. 이러한 리유로 객체가 무엇을 참고하는가 하는데 주의해야 한다.

2. 놓기

창문부품에 놓은 매체를 받아들일수 있게 하려면 그 창문부품에 대하여 구성자에서 setAcceptDrops(TRUE)을 호출하고 사건처리함수메쏘드들인 dragEnterEvent()와 dropEvent()를 재정의한다. 더 복잡한 응용프로그람들에서도 dragMoveEvent()와 dragLeaveEvent()의 재정의가 필요하다.

실례로 본문과 화상 놓기를 받아들이기 위하여

```
MyWidget::MyWidget(...) :   QWidget(...)
{
  ...
  setAcceptDrops(TRUE);
}

void MyWidget::dragEnterEvent(QDragEnterEvent* event)
{
  event->accept( QTextDrag::canDecode(event) ||
                      QImageDrag::canDecode(event) );
```

```
}

void MyWidget::dropEvent(QDropEvent* event)
{
  QImage image;
  QString text;

  if ( QImageDrag::decode(event, image) ) {
    insertImageAt(image, event->pos());
  } else if ( QTextDrag::decode(event, text) ) {
    insertTextAt(text, event->pos());
  }
}
```

3. 오려둠판

QDragObject와 QDragEnterEvent, QDragMoveEvent, QDropEvent클라스들은 모두 형정보를 제공하는 객체들의 클라스 QMimeSource의 파생클라스들이다. QDragObject에 기초하여 자료를 전송한다면 끌어다놓기를 얻을뿐아니라 무료로 자르기 및 붙이기도 얻을수 있다. QClipboard는 두개 함수를 가진다.

```
setData(QMimeSource*)
QMimeSource* data()const
```

이 함수들을 리용하여 보통 오려둠판에 끌어다놓기정보를 넣을수 있다.

```
void MyWidget::copy()
{
  QApplication::clipboard()->setData(
    new QTextDrag(myHighlightedText()) );
}

void MyWidget::paste()
{
  QString text;
  if ( QTextDrag::decode(QApplication::clipboard()->data(), text) )
    insertText( text );
}
```

지어 QDragObject파생클라스들을 파일입출력의 부분으로 사용할수 있다. 실례로 자기 응용프로그람이 CAD설계를 DXF형식으로 부호화하는 QDragObject의 파생클라스를 가지고있으면 보관과 적재코드는 다음과 같다.

```
void MyWidget::save()
{
  QFile out(current_file_name);
  if ( out.open(IO_WriteOnly) ) {
    MyCadDrag tmp(current_design);
    out.writeBlock( tmp->encodedData( "image/x-dxf" ) );
  }
}
```

```
void MyWidget::load()
{
  QFile in(current_file_name);
  if ( in.open(IO_ReadOnly) ) {
    if ( !MyCadDrag::decode(in.readAll(), current_design) ) {
      QMessageBox::warning( this, "Format error",
        tr("The file \"%1\" is not in any supported format").arg(current_file_name)
      );
    }
  }
}
```

QDragObject파생클라스는 MyDxfDrag가 아니라 MyCadDrag이다. 왜냐하면 앞으로 그것을 확장하여 다른 응용프로그람들에 DXF, DWG, SVF, WMF, 혹은 QPicture자료를 제공할수 있기때문이다.

4. 끌어다놓기작용

더 간단한 경우에 끌어다놓기의 목표는 끌고있는 자료의 사본을 받아들이고 원본을 삭제하겠는가를 결정한다. 이것은 QDropEvent에서의 Copy작용이다. 또한 목표는 다른 작용 특히 Move와 Link작용을 리해하도록 선택할수 있다. 목표는 Move작용을 리해하고 복사와 삭제조작들에 둘다 응답할수 있으며 원천은 자료자체를 삭제하려고 시도하지 않는다. 목표가 Link를 리해하면 그것은 원시정보에 대한 그자체의 참고를 보관하고 다시 원천은 원본을 삭제하지 않는다. 끌어다놓기작용의 가장 일반적인 사용은 같은 창문부품안에서 Move를 수행할 때이다.

끌기작용의 다른 주요한 사용은 본문 및 uri목록과 같은 참고형을 리용하는 경우이다. 여기서 끌기하는 자료는 실제로 파일들이나 객체들에로의 참고이다.

5. 새로운 끌어다놓기형의 추가

우의 DXF실례에서 제안한것처럼 끌어다놓기는 본문과 화상에만 제한되지 않는다. 임의의 정보를 끌어다놓을수 있다. 응용프로그람들사이에서 정보를 끌기하려면 응용프로그람들은 받아들이고 생성할수 있는 자료형식들을 서로 알릴수 있게 하여야 한다. 이것은 MIME형들에 의해 달성된다. 즉 끌기원천은 생성할수 있는 MIME형들의 목록(제일 적합한것으로부터 적합하지 않은 순서로)을 제공하고 놓기목표는 받아들일수 있는것이 어느것인가 선택한다. 실례로 QTextDrag는 text/plain의 MIME형(보통 비형식화된 본문)과 유니코드형식들인 text/utf16와 text/utf8에 대한 지원을 제공한다. QImageDrag는 image/*에 제공되는데 여기서 *는 QImageIO가 지원하는 화상형식이다. QUriDrag파생클라스는 파일이름 (혹은 URL)들의 목록을 전송하기 위한 표준형식인 text/uri-list를 제공한다.

QDragObject파생클라스를 사용할수 없는 형의 정보에 대한 끌어다놓기를 실현하는데서 가장 중요한 첫단계는 적합한 현존형식을 찾는것이다. IANA(Internet Assigned Numbers Authority)는 ISI(Information Sciences Institute)에서 MIME매체형들의 계층목록을 제공한다. 표준MIME형들의 사용은 현재와 앞으로 자기 응용프로그람과 다른 쏘프트웨어와의 호상운영성을 최대화한다.

추가적인 매체형을 지원하려면 QDragObject 혹은 QStoredDrag의 파생클라스를 만든다. 여러매체형에 대한 지원을 제공할 필요가 있을 때 QDragObject의 파생클라스를 만든다. 어떤 형이 충분할 때 더 간단한 QStoredDrag의 파생클라스를 만든다.

QDragObject의 파생클라스들은 const char* format(int i) const와 QByteArray encodedData(const char* mimetype) const성원들을 재정의하며 매체자료를 부호화하기 위한 set메쏘드, 들어오는 자료를 복호화하기 위한 정적함수들인 canDecode()와 decode(), QImageDrag의 bool canDecode(QMimeSource*) const와 QByteArray decode(QMimeSource*) const를 제공한다. 물론 이 메쏘드들중 일부를 생략하여 어떤 매체형에 대하여 끌기만 혹은 놓기만 제공할수 있다.

QStoredDrag의 파생클라스들은 매체자료를 부호화하기 위한 set메쏘드와 들어오는 자료를 복호화하기 위한 같은 정적성원들인 canDecode()와 decode()를 제공한다.

6. 고급한 끌어다놓기

오려둠판모형에서 사용자는 원천정보를 자르거나 복사하고 후에 그것을 붙이기할수 있다. 마찬가지로 끌어다놓기모형에서 사용자는 정보의 사본을 끌기할수 있으며 혹은 정보자체를(이동하여) 새 위치에 끌기할수 있다. 그러나 끌어다놓기모형은 프로그람작성자에게 추가적인 복잡성을 제기한다. 프로그람은 놓기(붙이기)가 수행될 때까지 사용자가 자르려는지 복사하려는지 모른다. 응용프로그람들사이의 끌기에는 차이가 없지만 응용프로그람안에서 끌기하는 경우에는 주의해야 한다. 실례로 문서에서 주위의 본문을 끌기위하여 끌기시작점과 놓기사건은 다음과 같이 만들수 있다.

```
void MyEditor::startDrag()
{
  QDragObject *d = new QTextDrag(myHighlightedText(), this);
  if ( d->drag() && d->target() != this )
    cutMyHighlightedText();
}

void MyEditor::dropEvent(QDropEvent* event)
{
  QString text;

  if ( QTextDrag::decode(event, text) ) {
    if ( event->source() == this && event->action() ==
       QDropEvent::Move ) {
      // Careful not to tread on my own feet
      event->acceptAction();
      moveMyHighlightedTextTo(event->pos());
    } else {
      pasteTextAt(text, event->pos());
    }
  }
}
```

일부 창문부품들은 자료를 그 우로 끌기할 때 yes 혹은 no응답보다 더 고유하다. 실례로 CAD프로그람은 오직 보기에서 본문객체들에로 본문의 놓기를 받아들일수 있다. 이 경우에 dragMoveEvent()가 사용되고 끌기를 받아들이거나 무시하는 구역이 주어진다.

```
void MyWidget::dragMoveEvent(QDragMoveEvent* event)
{
  if ( QTextDrag::canDecode(event) ) {
```

```
    MyCadItem* item = findMyItemAt(event->pos());
    if ( item )
      event->accept();
  }
}
```

객체들을 찾는 계산이 아주 느리고 접수를 허용하는 구역을 체계에 알려주어 개선된 성능을 얻을수 있다.

```
void MyWidget::dragMoveEvent(QDragMoveEvent* event)
{
  if ( QTextDrag::canDecode(event) ) {
    MyCadItem* item = findMyItemAt(event->pos());
    if ( item ) {
      QRect r = item->areaRelativeToMeClippedByAnythingInTheWay();
      if ( item->type() == MyTextType )
        event->accept( r );
      else
        event->ignore( r );
    }
  }
}
```

또한 dragMoveEvent()는 시계를 기동하기 위하여, 창문을 흘림하기 위하여, 혹은 무엇인가 적당한 일을 하기 위하여 끌기진척과정으로서 시각적인 반결합을 줄 필요가 있는 경우에 사용될수 있다. (그러나 dragLeaveEvent()에서 흘림과 시계를 중지하여야 한다.)

또한 QApplication객체(qApp대역변수로서 사용할수 있다.)는 다음 함수들과 관련한 끌어다놓기를 제공한다. 즉 QApplication::setStartDragTime(), QApplication::setStartDragDistance(), 그리고 그에 대응하는 얻기함수들, QApplication::startDragTime(), QApplication::startDragDistance().

7. 다른 응용프로그람들과의 호상운영

X11에서는 공개XDND통신규약을 사용하고 Windows에서 Qt는 OLE표준을 사용하며 Qt/Mac는 Carbon Drag Manager를 사용한다. X11에서는 XDND가 MIME를 사용하므로 번역이 필요없다. Qt API는 가동환경에 관계없이 같다. Windows에서 MIME-를 인식하는 응용프로그람들은 MIME형의 오려둠판형식이름을 리용하여 교제할수 있다. 이미 일부 Windows응용프로그람들은 오려둠판형식들에 MIME명명관례를 사용한다. 내적으로 Qt는 MIME형들에 대하여 소유권이 있는 오려둠판형식들을 변환하기 위한 편의도구들을 가지고있다. 이 대면부는 어떤 경우에는 공개로 만들어지지만 지금 그러한 변환을 해야 한다면 Qt의 기술봉사를 받아야 한다.

또한 X11에서 Qt는 Motif Drag&Drop Protocol를 거쳐서 놓기를 유지한다. 그 실현은 원래 Daniel Dardailler에 의해 작성되였고 Matt Koss와 Trolltech에 의해 Qt에 받아들인 코드와 결합되였다.

제6절. 사건과 사건려과기

Qt에서 사건은 QEvent를 계승하는 객체이다. 사건들은 QObject::event()호출을 통하여 QObject를 계승하는 객체들에 배달된다. 사건배달은 사건이 발생하였고 QEvent가 정확히 무엇인가 가리키며 QObject가 응답할 필요가 있다는것을 의미한다. 대부분의 사건들은 QWidget와 그 파생클라스들에 고유하지만 도형처리와 관련하지 않은 중요한 사건들(실례로 QSocketNotifier에 의하여 사용되는 사건인 소케트기능)이 있다.

일부 사건들은 창문체계로부터 일어나며(실례로 QMouseEvent) 일부 사건들은 다른 원천(실례로 QTimerEvent)으로부터 일어나며 일부는 응용프로그람으로부터 일어난다. 보통 Qt는 균형이 이루어져있으므로 Qt자체의 사건순환고리가 수행하는것과 같은 방법으로 정확히 사건들을 보낼수 있다.

대부분의 사건형은 특수한 클라스들이다. 가장 일반적인것들은 QResizeEvent와 QPaintEvent, QMouseEvent, QKeyEvent, QCloseEvent이다. 다른것들도 많은데 약 40여개 된다.

매개 클라스는 QEvent의 파생클라스이고 사건에 고유한 기능들을 추가한다. 실례로 QResizeEvent. QResizeEvent의 경우에 QResizeEvent::size()와 QResizeEvent::oldSize()가 추가된다.

일부 클라스들은 하나이상의 사건형을 유지한다. QMouseEvent는 마우스이동과 누르기, Shift누르기, 끌기, 찰칵, 오른단추누르기 등을 유지한다.

프로그람들이 여러가지 방식으로 반응해야 하므로 Qt의 사건배달기구는 유연해야 한다. QApplication::notify()는 응용프로그람에 대하여 간단히 설명한다.

사건을 배달하는 표준방법은 가상함수를 호출하는것이다. 실례로 QPaintEvent는 QWidget::paintEvent()호출에 의하여 배달된다. 이 가상함수는 적당히 반응하여 보통 창문부품을 다시 그리기하여 응답할수 있다. 가상함수의 실현에서 필요한 작업을 모두 수행하지 않으면 기초클라스의 실현을 호출할 필요가 있다. 실례로

```
MyTable::contentsMouseMoveEvent( QMouseEvent *me )
{
  // my implementation
  QTable::contentsMouseMoveEvent( me ); // 그것을 넘겨준다
}
```

기초클라스의 함수를 교체하려면 자체로 모든것을 실현해야 하지만 기초클라스의 기능을 확장하려고만 한다면 요구되는것을 실현하고 기초클라스를 호출한다.

때때로 그러한 사건에 고유한 함수가 없거나 사건에 고유한 함수가 충분하지 않다. 가장 일반적인 실례는 건누르기이다. 보통 건누르기는 QWidget에 의해 해석되여 건반초점을 이동시키지만 일부 창문부품들은 타브건을 요구한다.

이 객체들은 일반사건처리함수인 QObject::event()을 재정의할수 있으며 일반적인 처리의 전후에 사건처리를 수행하거나 사건처리를 완전히 교체한다. 타브를 해석하며 응용프로그람에 고유한 사용자정의사건을 가지는 매우 비일반적인 창문부품을 포함할수 있다.

```
bool MyClass:event( QEvent *evt ) {
  if ( evt->type() == QEvent::KeyPress ) {
    QKeyEvent *ke = (QKeyEvent *)evt;
    if ( ke->key() == Key_Tab ) {
      // special tab handling here
      ke->accept();
```

```
        return TRUE;
      }
    } else if ( evt->type() >= QEvent::User ) {
      QCustomEvent *ce = (QCustomEvent*) evt;
      // custom event handling here
      return TRUE;
    }
    return QWidget::event( evt );
  }
```

일반적으로 객체는 다른 객체들의 사건을 고찰할 필요가 있다. Qt는 QObject::installEventFilter()에 의하여 이것을 지원한다.(그리고 대응하는것은 삭제한다.) 실례로 보통 대화칸은 일부 창문부품들에서 건누르기를 려파하려고 한다. 실례로 Return건처리를 수정하려고 한다.

사건려파기는 목표객체가 수행하기전에 사건들을 처리하게 한다. 려파기의 QObject::eventFilter()실현이 호출되고 려파기를 받아들이거나 거부하고 사건의 앞으로의 처리를 허용하거나 거부한다. 모든 사건려파기들이 앞으로의 사건처리를 허용하면 사건은 목표객체자체에 보내진다. 그중 하나가 처리를 중지하면 목표와 그후의 사건려파기들은 사건을 전혀 보지 못하게 한다.

또한 QApplication에 사건려파기를 설치함으로써 전체 응용프로그람을 위한 사건들을 모두 려파할수도 있다. 이것은 마우스와 건반동작을 모두 조사하기 위하여 QToolTip가 수행하는것이다. 이것은 아주 강력하지만 전체 응용프로그람에서 모든 단일사건의 사건송달이 떠지게 하므로 피하는것이 제일 좋다.

대역사건려파기들은 객체에 고유한 려파기들보다 먼저 호출된다.

끝으로 많은 응용프로그람들은 자체의 사건들을 창조하고 송신하려고 한다.

기본형의 사건창조는 아주 단순하다. 즉 관련한 형의 객체를 창조하고 QApplication::sendEvent() 혹은 QApplication::postEvent()를 호출한다.

sendEvent()는 사건을 즉시 처리한다. sendEvent()가 되돌아올 때 (사건려파기와) 객체는 이미 사건에 의하여 처리된다. 많은 사건클라스들에 대하여 호출된 마지막 처리함수에 의하여 사건을 받아들였는가 거부하였는가를 결정하는 isAccepted()라는 함수가 있다.

postEvent()는 후의 발송하기 위한 대기렬에 사건을 기입한다. 다음번에 Qt의 기본사건순환고리를 실행하고 기입된 사건들을 최적화하여 모두 발송한다. 실례로 여러개의 크기조절사건들이 있으면 그것들은 하나로 압축된다. 같은것을 그리기사건들에도 적용한다. 즉 QWidget::update()는 깜빡거림을 최소화하고 여러번의 재그리기를 피하여 속도를 증가시키는 postEvent()를 호출한다.

또한 postEvent()는 흔히 객체를 초기화할 때 리용되므로 기입된 사건은 전형적으로 객체의 초기화가 완료된 후에 곧 발송된다.

사용자정의사건들을 창조하려면 QEvent::User보다 큰 사건번호를 정의할 필요가 있으며 자기의 사용자정의사건에 대한 특성을 넘기기 위하여 QCustomEvent의 파생클라스를 만들 필요가 있다(QCustomEvent문서를 참고).

제7절. 건반초점

Qt의 창문부품들은 GUI에서 습관된 방법으로 건반초점을 처리한다.

기본문제는 사용자의 건치기가 화면우의 여러 창문들중 임의의것과, 고찰하는 창문안의 여러 창문부품들중 하나에 보내질수 있는것이다. 체계는 건치기가 전달되는것이 어느 응용프로그람, 그 응용프로그람의 어느 창문, 그 창문의 어느 창문부품인가를 결정해야 한다.

1. 초점이동

특별한 창문부품에로 건반초점을 보내는 방법은 다음과 같다.

① 사용자가 Tab(또는 Shift+Tab 혹은 Enter)를 누른다.

② 사용자가 창문부품을 선택한다.

③ 사용자가 건반지름건을 누른다.

④ 사용자가 마우스바퀴를 사용한다.

⑤ 사용자가 창문에로 초점을 이동하고 응용프로그람은 창문안의 어느 창문부품이 초점을 얻어야 하는가를 결정한다.

매 방법에 대하여 고찰하자.

(1) Tab 혹은 Shift+Tab건을 누른다.

Tab누르기는 건반을 리용하여 초점을 옮기는 가장 일반적인 방법이다. 흔히 자료입력응용프로그람들에서 Enter는 Tab와 같은 일을 한다.

Tab누르기는 일반적으로 사용하는 모든 창문에서 초점이 다음 창문부품으로 이동하게 한다. Tab는 순회목록의 한 방향으로 초점을 이동시키고 Shift+Tab는 다른 방향으로 이동시킨다. Tab를 누를 때 한 창문부품으로부터 다른 창문부품으로 이동하는 순서를 타브순서라고 한다.

Qt에서 이 목록은 QFocusData클라스에 보관된다. 창문마다 하나의 QFocusData객체가 있으며 QWidget::setFocusPolicy()가 적당한 QWidget::FocusPolicy에 의해 호출될 때 창문부품들은 자동적으로 목록의 끝에 추가된다. QWidget::setTabOrder()를 사용하여 타브순서를 사용자가 정의할수 있다. (그렇게 하지 않으면 Tab는 일반적으로 초점을 창문부품을 구성한 순서로 이동시킨다.) Qt Designer는 타브순서를 시각적으로 변경하는 수단을 제공한다.

Tab누르기가 아주 일반적이므로 초점을 가질수 있는 대부분의 창문부품들은 타브초점을 유지해야 한다. 그러나 드문히 쓰이는 창문부품들과 건반지름건이 있는 창문부품으로 초점을 옮기는 오유처리함수는 례외이다.

실례로 자료입력대화칸에서 모든 경우의 1%만 필요한 마당이 있을수 있다. 그러한 대화칸에서 Tab는 이 마당을 건너뛸수 있고 대화칸은 다음 기구들중 하나를 사용할수 있다.

① 프로그람이 마당이 요구되는지 결정할수 있다면 사용자가 입력을 끝내고 OK를 누를 때 혹은 사용자가 다른 마당에 대한 조작을 끝낸 후 Enter를 누를 때 거기로 초점을 옮길수 있다. 또는 마당을 타브순서에 포함하지만 그것을 허용하지 않거나 사용자가 다른 마당들에서 설정한 보기에 적당하면 그것을 허용한다.

② 마당의 표식은 이 마당에로 초점을 옮기는 건반지름건을 포함할수 있다.

Tab기능에 관한 다른 례외는 타브의 삽입을 유지해야 하는 본문입력창문부품들이다. 거의 모든 본문편집기는 이 부류에 속한다. Qt는 Control+Tab를 Tab로서, Control+Shift+ Tab를 Shift+Tab로서 취급하며 그러한 창문부품들은 QWidget::event()를 재정의하고 QWidget::event()를 호출하기전에 Tab를 처리하여 다른 모든 건들의 표준처리를 얻는다. 그러나 일부 체계가 Control+Tab를 다른 목적에

사용하고 많은 사용자들이 Control+Tab를 모르므로 이것은 완전한 해결책이 아니다.

(2) 사용자가 창문부품을 마우스로 선택한다.

이것은 아마 콤퓨터들에서 마우스나 다른 위치지정장치를 리용하여 Tab를 누르는 것보다 더 일반적이다.

초점을 옮기기 위한 마우스누르기는 Tab보다 훨씬 더 강력하다. 초점을 창문부품으로 옮길 때 편집창문부품들에서는 마우스를 누른 곳으로 본문유표(창문부품의 내부초점)도 이동한다.

이것은 아주 일반적이고 사람들이 흔히 사용하므로 대부분의 창문부품들서 그것을 유지하는것이 좋다. 그러나 그것을 피하여야 할 중요한 리유도 있다. 즉 초점이 있는 창문부품으로부터 초점을 제거하려고 하지 않을수 있다.

실례로 문서처리프로그람에서 사용자가 B(강조)도구단추를 마우스로 선택할 때 건반초점에 어떤 일이 생기는가? 편집창문부품에서 초점이 있던 곳에 그대로 있어야 하는가 혹은 B단추로 초점을 옮겨야 하는가?

본문입력을 유지하는 창문부품들에서는 초점에 대한 누르기(click-to-focus)를 유지할것을 권고하고 마우스선택이 다른 효과를 가지는 창문부품들에서는 그것을 피할것을 권고한다. (또한 단추들에서는 건반지름건을 추가할것을 권고한다. QButton과 그 파생클라스들은 이것을 아주 쉽게 한다.)

Qt에서는 오직 QWidget::setFocusPolicy()함수가 초점에 대한 누르기에 영향을 준다.

(3) 사용자가 건반지름건을 누른다.

건반지름건들에 대하여 초점을 옮기는것은 례외적인 일이다. 이것은 이행금지대화칸을 펼침으로써 암시적으로 발생시킬수 있으나 QLabel::setBuddy()와 QGroupBox, QTabBar에 의해 제공되는것들과 같은 초점지름건들을 사용하여 명시적으로 발생시킬수 있다.

사용자가 이행하려고 할수 있는 모든 창문부품들에 대하여 지름건초점을 유지할것을 권고한다. 실례로 Alt+P를 누르면 인쇄대화칸이 나타나는것처럼 타브대화칸은 그 매개 페지에 대하여 건반지름건을 가진다. 그러나 이것이 지나쳐서는 안된다. 즉 오직 몇개의 건들만 리용하여 지령들에 건반지름건들을 제공하는것이 중요하다. 표준지름건목록에서 Alt+P는 Paste와 Play, Print, Print Here에도 사용될수 있다.

(4) 사용자가 마우스바퀴를 사용한다.

Microsoft Windows에서 마우스바퀴의 사용은 늘 건반초점을 가지는 창문부품에 의해 처리된다. Mac OS X와 X11에서는 마우스사건들을 얻는 다른 창문부품에 의하여 처리된다.

Qt가 가동환경의 이러한 차이를 처리하는 방법은 바퀴를 사용할 때 창문부품들이 건반초점을 옮기게 하는것이다. 매개 창문부품에 대하여 정확한 초점정책을 실시하여야 응용프로그람들이 Windows와 Mac OS X, X11에서 정확히 작업할수 있다.

(5) 사용자가 창문에로 초점을 이동한다.

이 상황에서 응용프로그람은 창문안의 어느 창문부품이 초점을 받아들여야 하는가를 결정해야 한다.

이것은 간단하다. 즉 초점이 이전에 이 창문안에 있었다면 초점을 가져야 할 마지막 창문부품이 그것을 도로 찾아야 한다. Qt는 이것을 자동적으로 수행한다.

초점이 이전에 이 창문안에 없었고 초점이 어디서 시작해야 하는지 안다면 QWidget::show()하기전에 초점을 받아들여야 하는 창문부품에 대하여 QWidget::setFocus()를 호출한다. 그렇게 하지 않으면 Qt는 적당한 창문부품을 선택한다.

제8절. 표준지름건

응용프로그람들은 언제나 작용과 련관된 지름건을 정의해야 한다. Qt는 지름건을 충분히 지원한다. 실례로 QAccel::shortcutKey()를 들수 있다.

여기에 지름건에 대한 마이크로쏘프트의 권고와 Open Group의 권고안이 있다. 대다수의 지령들에서 Open Group는 마이크로쏘프트의 권고에 동의하지 않는다.

강조체문자+Alt는 마이크로쏘프트사의 권고선택안이다. 실례로 Apply단추는

```
QButton::setText( tr("&Apply") );
```

이다.

모순되는 지령(실례로 About와 Apply)이 있으면 자체로 결심하여 정한다.

About
Always on **T**op
Apply
Back
Browse
Close (CDE: Alt+F4; Alt+F4)은 창문닫기
Copy (CDE: Ctrl+C, Ctrl+Insert)
Copy Here
Create **S**hortcut
Create **S**hortcut Here
Cu**t**
Delete
Edit
Exit (CDE: E**x**it)
Explore
File
Find
Help
Help **T**opics
Hide
Insert
Insert **O**bject
Link Here
Ma**x**imize
Mi**n**imize
Move
Move Here
New
Next
No
Open
Open **W**ith
Page Set**u**p
Paste
Paste **L**ink
Paste **S**hortcut
Paste **S**pecial
Pause
Play
Print
Print Here
P**r**operties
Quick View
Redo (CDE: Ctrl+Y, Shift+Alt+Backspace)
Repeat
Restore
Resume
Retry
Run
Save
Save **A**s
Select **A**ll
Se**n**d To
Show
Size
S**p**lit
Stop
Undo (CDE: Ctrl+Z or Alt+Backspace)
View
What's This?
Window
Yes

또한 다른 많은 건들과 작용(Alt가 아니라 다른 수식건을 리용한다.)들이 있다.

제9절. Qt Netscape플라그인확장

Qt Netscape Plugin쏘프트웨어는 Netscape, Mozilla 및 Netscape의 LiveConnect통신규약을 유지하는 다른 웨브열람기들에서 Unix/Linux 및 MS-Windows상에서 사용할수 있는 열람기플라그인들을 작성하기 쉽게 한다. 현대판의 MSIE는 이 통신규약을 유지하지 않는다. ActiveQt틀거리를 사용하여 이 열람기용플라그인을 개발한다.

Netscape플라그인확장은 다음의 클라스들로 이루어진다.

- QNPlugin
- QNPInstance
- QNPWidget
- QNPStream

1. 방법

① Netscape로부터 플라그인 SDK를 내리적재하고 거기에서 $QTDIR/extensions/ nsplugin/src에 다음의 파일들을 복사한다.

- common/npwin.cpp
- common/npunix.c
- include/npapi.h
- include/npupp.h
- include/jri.h
- include/jri_md.h
- include/jritypes.h

② 자기 Qt배포물의 extensions/nsplugin/src 등록부에 있는 Netscape Plugin 확장서고를 구축한다. 이것은 자기의 플라그인코드와 련결될 정적서고를 산출한다.

③ 플라그인클라스문서를 읽고 실례플라그인들을 시험한다.

④ 대부분의 개발을 독자적인 Qt응용프로그람으로서 수행한다. Netscape Plugins의 오유수정은 시끄럽다. 자기 플라그인에서 signal(2)를 사용하여 자기 열람기가 핵심부출력을 금지한다면 그 출력을 가능하게 할수 있다.

다음에 가동환경에 고유한 구축단계들을 설명한다.

플라그인에 의해 표시된 파일들이 HTTP봉사기(http://... URL을 사용하여)에 의해 제공된다면 봉사기는 가령 Apache의 mime.types파일을 편집함으로써 그 파일의 정확한 MIME형을 전송하도록 환경구성되여야 한다. 파일들이 //... URL을 거쳐서 표시되면 열람기는 파일이름확장자를 사용하여 파일형과 이로부터 적재할 플라그인을 결정한다. 사용자는 그 열람기가 애호하는 보조프로그람이나 응용프로그람부분에서 파일이름확장자를 설정할 필요가 있다.

1) X11하에서의 구축

- 실례의 Makefile들은 UNIX/X11에 적합하다.
- 사용자는 결과에 생기는 공유객체를 열람기의 Plugins등록부에 설치해야 한다.

2) Windows하에서의 구축

- Netscape플라그인과 작업하려면 Qt는 체계DLL경로에 있거나 정적서고로서 플라그인으로 콤파일되여야 한다.
- 플라그인들은 np*name*.dll로 명명되여야 하며 그렇지 않으면 열람기는 그것들을 무시한다.
- 련결단계는 다음과 같은것을 포함해야 한다.

/def:*name*.def

/dll
플라그인에 의해 받아들인 파일/MIME형을 정의하는 콤파일된 자원파일.
- 사용자는 열람기의 Plugins등록부에 결과로 생기는 DLL을 설치해야 한다.

2. 알려진 오유와 제한

Qt기초의 LiveConnect Plugin속박코드는 수많은 오유와 제한을 가지고있으나 많은 프로그람제품들에서 충분히 안정하다.
- 건반입력은 오직 2차창문(실례로 플라그인에 의해 창조된 대화칸)들에서 작업한다.
- 플라그인과 작업하려는 열람기사이의 양식성(modality)을 기대하지 말아야 한다.
- Unix/X11에서 Netscape 4.78은 모선오유로 완료되는 경향이 있다.
- Opaque크기조절동작은 열람기의 동작으로 인하여 산만스러워진다.

제10절. ActiveQt틀거리

이 모듈들은 Qt Enterprise Edition의 일부이고 무료(Free) 혹은 비상업판의 부분이 아니다.

Qt의 ActiveX는 Qt/Windows개발자들에게 다음과 같은 일을 가능하게 한다.

① Qt응용프로그람들에서 ActiveX봉사기에 의하여 제공된 ActiveX조종요소들을 호출하고 리용할수 있게 한다.

② ActiveX조종요소들과 같은 많은 량의 Qt창문부품들을 사용하여 그들의 Qt응용프로그람들을 ActiveX봉사기로서 쓸수 있게 한다.

1. ActiveQt모듈들

틀거리는 두개의 모듈로 이루어진다.

① QAxContainer

QAxContainer모듈은 QObject와 QWidget보조클라스들, COM객체들과 ActiveX조종요소들에 용기로서 작용하는 QAxObject와 QAxWidget를 실현하는 정적서고이다. 공유Qt서고에 대처하여 구축되였다면 QAxWidget는 Qt Designer에 창문부품플라그인으로서 통합된다.

또한 그 모듈은 Qt응용프로그람들에 매몰된 스크립트COM객체들에 Windows Script Host기술을 사용할수 있게 하는 클라스 QAxScript, QAxScriptManager과 QaxScriptEngine을 제공한다.

실례에는 Microsoft Internet Explorer를 매몰하는 웨브열람프로그람과 Microsoft Outlook와 내용을 동기화하는 주소록실례가 포함되여있다.

② QAxServer

QAxServer모듈은 프로쎄스내 및 실행가능형 COM봉사기용 기능을 실현하는 정적서고이다. 이 모듈은 QAxAggregated, QAxBindable 그리고 QAxFactory클라스들을 제공한다.

실례에는 .NET환경에서 그 객체들을 사용하는 련습은 물론 ActiveX조종요소들과 같은 각이한 QWidget보조클라스들을 제공하는 프로쎄스내 및 프로쎄스외봉사기들이 포함되여있다 .

2. ActiveQt도구

다음에 ActiveQt용도구프로그람들을 보여준다.

① DumpDoc도구

dumpdoc도구는 COM객체용 Qt형식문서를 생성하여 지정된 파일에 써넣는다.

표 4-5와 같은 지령행파라메터를 리용하여 dumpdoc를 호출한다.

표 4-5. DumpDoc지령행추가선택

추가선택	결과
-o *file*	출력을 *file*에 써넣는다.
object	*object*용문서를 생성한다.
-v	판정보를 인쇄한다.
-h	방조를 인쇄한다

*object*는 국부콤퓨터에 설치된 객체이고(즉 원격객체들은 지원하지 않는다) 속성을 통하여 호출할수 있는 보조객체들을 포함하여야 한다. 즉 Outlook.Application/Session/ CurrentUser.

생성된 파일은 Qt문서형식을 리용하는 HTML파일일수 있다.

도구를 구축하려면 우선 QAxContainer서고를 구축해야 한다. 그다음 tools/dumpdoc에서 make도구를 실행한다.

② IDC도구

IDC도구는 ActiveQt구축체계의 부분으로써 불과 몇개행의 코드로 임의의 Qt2진파일을 완전COM객체봉사기로 전환한다.

IDC는 표4-6과 같은 지령행파라메터들을 리해한다.

표 4-6. IDC지령행추가선택

추가선택	결과
Dll -idl idl -version x.y	봉사기 dll의 IDL을 파일idl에 써넣는다. 형서고는 판 x.y를 가진다.
Dll -tlb tlb	dll 의 형서고를 tlb로 교체한다.
-v	판정보를 인쇄한다.
-regserver dll	Register the COM봉사기dll을 등록한다.
-unregserver	COM봉사기dll의 등록을 해제한다.

일반적으로 IDC를 수동적으로 호출할 필요가 없다면 qmake구축체계는 구축과정에 필요한 발송(post)처리의 추가를 고려해야 한다(ActiveQt구축체계문서 참고).

③ ActiveX시험용기

이 응용프로그람은 ActiveX조종요소용 일반시험용기를 실현한다. 자기 체계에 설치된 ActiveX조종요소를 삽입하고 메쏘드를 실행하고 속성들을 수정할수 있다. 용기는 로그창문에서 오유수정출력은 물론 사건과 속성변경에 대한 정보를 기록할수 있다.

GUI는 QAxContainer모듈의 Qt Designer통합에 의하여 실현되였다. 코드부분은 Qt메타객체와 ActiveQt틀거리를 사용하며 응용프로그람코드에서 사용하지 않는것이 좋다.

응용프로그람을 리용하여 일정한 ActiveX의 실례를 작성할 때 QAxWidget클라스를 통하여 사용할수 있는 처리부와 신호, 속성들을 표시하고 자기가 실현한 ActiveX조종을 시험하거나 Qt응용프로그람에서 사용하려고 한다.

응용프로그람은 적재된 조종요소들을 프로그람작성하는데 JavaScript와 VBScript, Perl, Python으로 되여있는 스크립트파일들을 적재하고 실행할수 있다. QAxWidget2클라스를 사용하는 실레스크립트파일들은 scripts보조등록부에서 리용할수 있다.

이 실례의 qmake프로젝트는 판자원을 가진 자원파일 testcon.rc를 포함한다. 이것은 일부 ActiveX조종요소(실례로 Shockwave ActiveX조종요소)에 요구되며 그것은 그러한 판정보가 없으면 중단하거나 오동작할수 있다.

도구를 구축하려면 우선 QAxContainer서고를 구축해야 한다. 그다음

tools/testcon에서 make도구를 실행한 결과 만들어지는 testcon.exe를 실행한다.

제11절. Qt Motif확장

이 모듈은 Qt Enterprise판의 부분이며 무료 혹은 비상업판의 부분이 아니다.

1. 소개

Qt Motif확장은 낡은 Xt와 Motif기초응용프로그람들과 더 편리한 Qt도구일식의 통합을 방조한다. 이 확장은 Qt의 초기판들에 포함된 낡은 Xt/Motif Support Extension을 교체한다.

Qt Motif확장은 다음의 클라스들로 이루어진다.

- QMotif - Qt Motif확장의 기초를 제공한다.
- QMotifWidget - Motif창문부품용 QWidget API를 제공한다.
- QMotifDialog - Motif대화칸용 QDialog API를 제공한다.
- QXtWidget -이전의 Xt/Motif확장으로부터 Xt/Motif통합. 이 클라스는 유지되지 않고 수많은 문제들과 제한들이 알려져있다. 이것은 현존원천작업을 유지할 목적으로만 제공된다. 이것을 새 코드에서 사용하지 말아야 한다.

2. 일반문제

① 부정확한 CDE색구조

QMotifWidget와 QMotifDialog는 QApplication가 사용하는것과 같은 시각적인 색략도와 색깊이를 사용한다. CDE를 사용할 때 색구조는 기정이 아닌 시각적인 색략도와 색깊이를 사용할 때 정확하지 않을수 있다. 이 문제를 처리하려면 기동파일($HOME/.dt/sessions/current/dt.resources)에 다음과 같은 자원문자렬을 추가하여야 한다.

```
*userColorObj: false
```

② X11머리부충돌

X11머리부들은 Qt머리부들과 충돌하는 일부 상수들을 정의한다. 해결책은 우선 Qt머리부들을 포함하고 뒤이어 이 확장의 머리부들을 포함하고 다음에 모든 Xt/Motif와 X11머리부들을 포함하는것이다. 실례로

```
// Qt headers first
#include <qapplication.h>
#include <qpushbutton.h>
#include <qsocket.h>
...

// QMotif* headers next
#include <qmotif.h>
#include <qmotifdialog.h>
#include <qmotifwidget.h>

// Xt/Motif and X11 headers last
#include <X11/Xlib.h>
#include <Xt/Intrinsic.h>
#include <Xm/Xm.h>
...
```

③ 여러화면지원

QMotifWidget를 QDesktopWidget와 함께 사용하여 여러 화면에 제일 웃준위창문들을 창조할수 있다. 일반적인 오유는 비기정화면우에 QMotifWidget를 창조하고 Xt/Motif창문부품들을 기정화면우에 창조하는것이다. 해결책은 QMotifWidget와 Xt/Motif자식을 둘다 지정하는것이다. 실례로

```
Display *dpy = QApplication::desktop()->x11Display();
Arg args[1];
// QMotifWidget와 XmMainWindow이 둘다 화면1에 있다는것을 확인한다
XtSetArg(args[0], XtNscreen, ScreenOfDisplay(dpy, 1));
QMotifWidget *toplevel = new QMotifWidget(QApplication::desktop()->screen(1),
    xmMainWindowWidgetClass, args, 1, "mainwindow");
```

3. QMotif기능확장

다음의 실례프로그람들은 낡은 Xt와 Motif기초코드를 더 편리한 Qt도구일식에로 통합과정을 방조하는 QMotif의 사용법을 설명한다.

1) Motif사건순환고리의 사용

머리부파일

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <qmainwindow.h>

class QMotifWidget;

class MainWindow : public QMainWindow
{
public:
  MainWindow();

private:
  QMotifWidget *customwidget;
};

#endif // MAINWINDOW_H
```

실현파일

```
#include "mainwindow.h"
#include <qapplication.h>
#include <qmotif.h>

int main( int argc, char **argv )
{
  XtSetLanguageProc( NULL, NULL, NULL );

  QMotif integrator( "customwidget" );
  QApplication app( argc, argv );
```

```
  MainWindow mainwindow;
  app.setMainWidget( &mainwindow );
  mainwindow.show();

  return app.exec();
}

#include "mainwindow.h"
#include <qapplication.h>
#include <qmenubar.h>
#include <qpopupmenu.h>
#include <qstatusbar.h>
#include <qmotifwidget.h>
#include <Xm/Form.h>
#include <Xm/PushB.h>
#include <Xm/Text.h>

MainWindow::MainWindow() : QMainWindow( 0, "mainwindow" )
{
  QPopupMenu *filemenu = new QPopupMenu( this );
  filemenu->insertItem( tr("&Quit"), qApp, SLOT(quit()) );

  menuBar()->insertItem( tr("&File"), filemenu );
  statusBar()->message( tr("This is a QMainWindow with an XmText widget.") );

  customwidget = new QMotifWidget( this, xmFormWidgetClass, NULL, 0, "form" );

  XmString str;
  Arg args[6];

  str = XmStringCreateLocalized( "Push Button (XmPushButton)" );
  XtSetArg( args[0], XmNlabelString, str );
  XtSetArg( args[1], XmNleftAttachment, XmATTACH_FORM );
  XtSetArg( args[2], XmNrightAttachment, XmATTACH_FORM );
  XtSetArg( args[3], XmNbottomAttachment, XmATTACH_FORM );
  Widget button =
    XmCreatePushButton( customwidget->motifWidget(), "Push Button", args, 4 );
  XmStringFree( str );

  XtSetArg( args[0], XmNeditMode, XmMULTI_LINE_EDIT );
  XtSetArg( args[1], XmNleftAttachment, XmATTACH_FORM );
  XtSetArg( args[2], XmNrightAttachment, XmATTACH_FORM );
  XtSetArg( args[3], XmNtopAttachment, XmATTACH_FORM );
  XtSetArg( args[4], XmNbottomAttachment, XmATTACH_WIDGET );
  XtSetArg( args[5], XmNbottomWidget, button );
```

```
  Widget texteditor =
    XmCreateScrolledText( customwidget->motifWidget(), "Text Editor", args, 6 );

  XtManageChild( texteditor );
  XtManageChild( button );

  setCentralWidget( customwidget );

  resize( 400, 600 );
}
```

2) Motif대화칸

머리부파일

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H
#include <qmotifwidget.h>

class MainWindow : public QMotifWidget
{
public:
  MainWindow();
  void showMotifDialog();
  void showQtDialog();
};

#endif // MAINWINDOW_H
```

실현파일

```
#include <qapplication.h>
#include <qmotif.h>
#include "mainwindow.h"

int main( int argc, char **argv )
{
  XtSetLanguageProc( NULL, NULL, NULL );

  QMotif integrator( "dialog" );
  QApplication app( argc, argv );

  MainWindow mainwindow;
  app.setMainWidget( &mainwindow );
  mainwindow.show();

  return app.exec();
}

#include "mainwindow.h"
#include "dialog.h"
```

```
#include <Xm/MainW.h>
#include <Xm/RowColumn.h>
#include <Xm/CascadeB.h>
#include <Xm/PushB.h>
#include <Xm/PushBG.h>
#include <Xm/SeparatoG.h>
#include <Xm/Text.h>
#include <Xm/MessageB.h>
#include <Xm/Form.h>
#include <Xm/LabelG.h>

#include <qapplication.h>

static void motifDialogCallback( Widget, XtPointer client_data, XtPointer )
{
  MainWindow *mw = (MainWindow *) client_data;
  mw->showMotifDialog();
}

static void qtDialogCallback( Widget, XtPointer client_data, XtPointer )
{
  MainWindow *mw = (MainWindow *) client_data;
  mw->showQtDialog();
}

static void quitCallback( Widget, XtPointer client_data, XtPointer )
{
  MainWindow *mw = (MainWindow *) client_data;
  mw->close();
}

MainWindow::MainWindow()
  : QMotifWidget( 0, xmMainWindowWidgetClass, NULL, 0,
"mainwindow" )
{
  Widget menubar = XmCreateMenuBar( motifWidget(), "menubar", NULL, 0 );
  Widget filemenu = XmCreatePulldownMenu( menubar, "filemenu", NULL, 0 );
  Widget item;

  item = XtVaCreateManagedWidget( "Motif Dialog...",
            xmPushButtonGadgetClass, filemenu, XmNmnemonic, 'C', NULL );
  XtAddCallback( item, XmNactivateCallback, motifDialogCallback, this );

  item = XtVaCreateManagedWidget( "Qt Dialog...",
          xmPushButtonGadgetClass, filemenu, XmNmnemonic, 'Q', NULL );
  XtAddCallback( item, XmNactivateCallback, qtDialogCallback, this );
```

```
      item = XtVaCreateManagedWidget( "sep", xmSeparatorGadgetClass,
filemenu, NULL);

      item = XtVaCreateManagedWidget( "Exit",
              xmPushButtonGadgetClass, filemenu, XmNmnemonic, 'x', NULL );
      XtAddCallback( item, XmNactivateCallback, quitCallback, this );

      XmString str = XmStringCreateLocalized( "File" );
      item = XtVaCreateManagedWidget( "File", xmCascadeButtonWidgetClass,
menubar,XmNlabelString, str, XmNmnemonic, 'F', XmNsubMenuId,
filemenu, NULL );
      XmStringFree( str );

      Arg args[2];
      XtSetArg( args[0], XmNeditMode, XmMULTI_LINE_EDIT );
      Widget texteditor = XmCreateScrolledText( motifWidget(), "texteditor",
args, 1 );

      XtManageChild( menubar );
      XtManageChild( texteditor );

      // pick a nice default size
      XtVaSetValues( motifWidget(), XmNwidth, 400, XmNheight, 300,
NULL );

      setCaption( tr("QMotif Dialog Example") );
    }

    void MainWindow::showMotifDialog()
    {
      QMotifDialog dialog( this, "custom dialog", TRUE );
      dialog.setCaption( tr("Custom Motif Dialog") );

      Widget form = XmCreateForm( dialog.shell(), "custom motif dialog",
NULL, 0 );

      XmString str;
      Arg args[9];

      str = XmStringCreateLocalized( "Close" );
      XtSetArg( args[0], XmNlabelString, str );
      XtSetArg( args[1], XmNshowAsDefault, True );
      XtSetArg( args[2], XmNleftAttachment, XmATTACH_POSITION );
      XtSetArg( args[3], XmNleftPosition, 40 );
      XtSetArg( args[4], XmNrightAttachment, XmATTACH_POSITION );
```

```
    XtSetArg( args[5], XmNrightPosition, 60 );
    XtSetArg( args[7], XmNbottomAttachment, XmATTACH_FORM );
    XtSetArg( args[6], XmNtopOffset, 10 );
    XtSetArg( args[8], XmNbottomOffset, 10 );
    Widget button = XmCreatePushButton( form, "Close", args, 9 );
    XmStringFree( str );

    str = XmStringCreateLocalized( "This is a custom Motif-based dialog
using\n" "QMotifDialog with a QWidget-based parent." );
    XtSetArg( args[0], XmNlabelString, str );
    XtSetArg( args[1], XmNleftAttachment, XmATTACH_FORM );
    XtSetArg( args[2], XmNrightAttachment, XmATTACH_FORM );
    XtSetArg( args[3], XmNtopAttachment, XmATTACH_FORM );
    XtSetArg( args[4], XmNbottomAttachment, XmATTACH_WIDGET );
    XtSetArg( args[5], XmNbottomWidget, button );
    XtSetArg( args[6], XmNtopOffset, 10 );
    XtSetArg( args[7], XmNbottomOffset, 10 );
    Widget label = XmCreateLabelGadget( form, "label", args, 8 );
    XmStringFree( str );

    XtManageChild( button );
    XtManageChild( label );
    XtManageChild( form );

    XtAddCallback( button, XmNactivateCallback,
            (XtCallbackProc) QMotifDialog::acceptCallback, &dialog );

    dialog.exec();
  }

  void MainWindow::showQtDialog()
  {
    // custom Qt-based dialog using a Motif-based parent
    CustomDialog customdialog( motifWidget(), "custom dialog", TRUE );
    customdialog.exec();
  }
```

4. Qt Motif확장-이식단계

여기서는 Qt Motif확장을 리용하여 Motif에 기초한 프로그람을 Qt도구일식으로 완전히 이식하는 방법을 설명한다.

수십만행 지어는 수백만행의 코드를 가지는 큰 프로젝트를 한번에 이식하는 일이 드문히 있으며 그러한 시도는 아주 많은 개발자원을 요구하며 중요한 위험을 초래한다. 코드의 각 행은 다시 써야 할수 있고 각 사용자대면부를 다시 설계하고 질담보검사와 수속을 모두 다시 써야 할수 있다. Qt Motif확장은 매개의 개별적인 프로젝트와 유효자원에 적당한 증분이식을 위한 완전하고도 동작하는 대책을 제공한다. 사용자대면부와 관련

코드는 여러 걸음에 이식될수 있고 나머지 프로젝트는 전혀 변경할 필요가 없다.

이 설명은 현존Motif기초프로젝트를 Qt도구일식에 이식하는 프로그람작성자용이다. 여기서는 프로그람작성자들이 C/C++와 Xt/Motif에 대한 경험이 있으리라고 생각한다. 또한 Qt도구일식에 대한 경험도 필요하다.

-이 설명의 목적

이 설명의 목적은 Motif응용프로그람을 Qt도구일식에 통합하기 위한 충분한 지식을 독자들에게 제공하는것이다. 실례를 들어 설명한다. 실제의 Motif기초프로그람을 한걸음씩 전환한다. 이 걸음은 두가지 기술과 그 관계를 처리과정을 통하여 설명한다.

이식프로젝트로서 Motif 2.x배포물에 포함되는 todo시위프로그람을 사용한다. 이 설명은 extensions/motif/examples/walkthrough보조등록부에 포함되는 원천코드에로의 참고를 포함한다.

- 필수조건

Qt Motif Extension의 사용을 시작하기전에 다음의 요구를 만족시켜야 한다.

① X11R6.x와 Motif 2.x서고들을 사용하고있다.

② 프로젝트는 C++콤파일러와 호환될수 있다.

③ Qt Motif Extension을 구축하고 설치하였다.

④ 프로젝트를 Qt도구 및 Qt Motif Extension과 함께 구축 및 련결하였다.

X11R6.x과 Motif 2.x서고

Qt Motif Extension이 사용한 기구들은 X11R6출하물과 Motif 2.0출하물로부터 개발머리부와 서고들을 요구한다. 더 새로운 판을 사용할수도 있다.

C++콤파일러

Qt는 C++도구이므로 모든 새 코드는 C++로 씌여진다. 현존코드를 새 코드와 공존하기 위하여 C++콤파일러는 현존코드를 콤파일할수 있어야 한다.

현존코드와 새 코드를 따로따로 유지할수 있고 현존코드를 필요할 때 변환하거나 다시 쓸수 있다. 이것은 이식과정의 표준부분이고 이식을 시작하기전에 수행할 필요는 없다. 이것은 가장 일반적인 대본이고 현존 C코드를 이 설명에서 필요할 때 C++에 이식하여 보여준다.

Qt Motif확장의 구축 및 설치

Qt Motif Extension은 Qt도구와 함께 구축하고 설치되지 않는다. 이 확장은 extensions/motif 보조등록부에 있다. 이 동록부에서 make를 실행하여 확장과 실례들을 구축한다. 일단 확장이 구축되였으면 make install을 실행한다.

```
$ cd extensions/motif
$ make
$ make install
```

현재 Qt Motif Extension가 설치되고 사용할 준비가 되였다.

프로젝트를 Qt도구, Qt Motif확장을 가지고 구축 및 련결

간단히 qmake로 Makefile을 창조한다. -project선택은 qmake가 자동적으로 프로젝트파일을 생성하게 하나. qmake -project가 프로젝트파일을 생성한 다음 qmake를 다시 실행하여 Makefile을 생성한다. 이제는 make를 실행하여 프로젝트를 구축할수 있다.

```
$ qmake -project
$ qmake
$ make
```

정확히 구축되였지만 Motif서고와 련결하지 못하여 련결이 실패한다. 프로젝트파일에서 LIBS변수에 -lXm를 추가하여 qmake가 이것을 하게 한다. 이 프로젝트에서 Qt Motif Extension을 사용할 예정이므로 또한 서고목록에 -lqmotif를 추가해야 한다.

LIBS += -lXm -lqmotif

그러면 qmake 를 다시 실행하여 Makefile를 다시 생성하고 make로 다시 구축한다. 이번에 프로젝트는 성공적으로 련결하고 응용프로그람은 기대한대로 실행된다.

이제는 Qt Motif Extension을 리용할 준비가 되였다.

1) 시작하기

① QMotif와 QApplication로 시작

Qt를 사용하기 위하여 하나의 QApplication객체를 창조해야 한다. QApplication클라스는 사건의 모든 배포를 조종하고 다른 모든 Qt객체들과 창문부품들의 관리를 현시한다. QMotif클라스를 Qt Motif Extension으로부터 리용하여 QApplication과 XtAppContext이 공존하게 한다.

QApplication객체는 main()함수에서 창조되여야 한다. C++콤파일러에서 콤파일하도록 todo.c를 수정하여야 하므로 todo.c를 todo.cpp로 이름을 바꾼다.

다음으로 QMotif와 QApplication클라스용의 적당한 머리부를 추가한다.

다음으로 QMotif와 QApplication객체들을 창조한다. 외부XtAppContext을 가지고 QMotif를 창조하고 외부Display를 가지고 QApplication을 창조한다.

다음의 변경은 아직 필요없지만 Qt Motif Extension이 완전한 통합을 제공한다는것을 보여주기 위하여 포함된다. 보통 Motif기초프로그람은 XtAppMainLoop()함수를 리용하여 응용프로그람의 사건순환고리를 실행한다. 이것은 여전히 가능하지만 Qt도구에 이식하고있으므로 사건순환고리실행에 QApplication::exec()함수를 리용하는것이 좋다.

todo.c를 todo.cpp으로 이름을 바꾸었으므로 프로젝트파일을 변경하고 qmake를 재실행하여 Makefile을 다시 생성해야 한다. 프로젝트를 구축할 때 콤파일 및 련결오유들이 있으며 다음 절에서 이것들을 고찰한다.

② C++에로 이식

이 파일의 코드를 적당한 C++코드로 변환하여야 한다. 다행히도 변경은 그리 크지 않다. 현존C프로젝트들로부터 포함한 대부분의 파일들은 C++에 호환되지 않으므로 extern "C"블로크에 그것들을 포함하여 호환되게 만든다.

또한 앞방향선언되는 대역C함수들은 extern "C"블로크에 포함해야 한다.

manageCB()함수는 적당한 C++로 변환할 필요가 있다.

그리고 2개의 무효한 강제변환을 고착시켜야 한다. 하나는 Save()함수에 있다.

다른 무효한 강제변환은 Open()함수에 있다.

이 변경후에 프로젝트를 콤파일하고 련결하며 응용프로그람을 정확히 실행하고 연산한다.

2) 사용자대면부의 이식준비

프로젝트용 사용자대면부를 이식할 준비가 되였다. 소개에서 언급한것처럼 여러 걸음에 이것을 수행하여 응용프로그람이 여전히 이식과정의 매개 단계에서 사용할수 있도록 한다.

사용자대면부는 계층으로 고찰할수 있으며 매개 제일 웃준위창문과 대화칸은 부모(다른 제일웃준위창문이나 대화칸)에 의존한다. Motif XmMainWindow 창문부품을 리용할 때 이것은 2개의 개별적인 실체들로 고찰되여야 한다. 즉 올리펼침차림표계층을 가진 기본창문창문부품과 기본창문창문부품에 포함된 보기창문부품. 보기창문부품은 기본창문창문부품에 의존한다.

의존관계를 가지지 않는 부분들을 이식하기 시작하며 모든 부분을 이식할 때까지 아주 훌륭한 의존관계나무와 작업한다.

프로젝트의 계층은 다음과 같다.

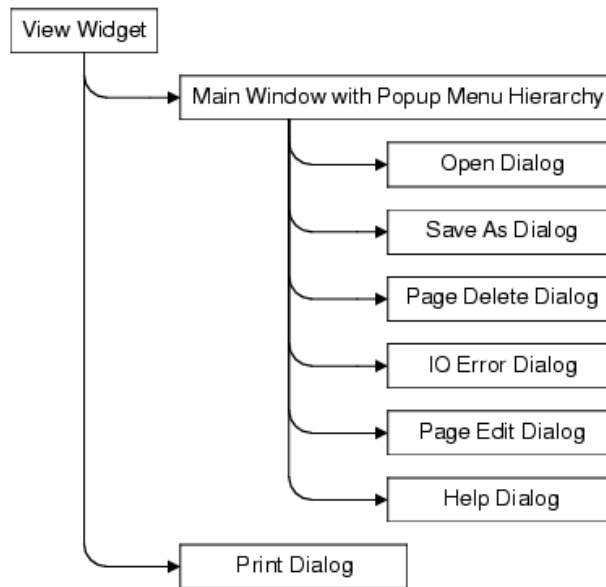



그림 4-1. 프로젝트의 계층

모든 대화칸, 다음에 올리펼침차림표계층을 가지는 기본창문창문부품, 끝으로 보기창문부품의 이식과 교체를 한다.

그러나 아직 시작할 준비가 안되였다. 이식과정은 오랠수 있지만 어렵지는 않다. Open, Save As, Page Delete 그리고 IO Error대화칸들은 Motif XmFileSelectionBox과 XmMessageBox대화칸을 사용한다. Qt는 즉시 리용할수 있는 QFileDialog와 QMessageBox와 류사한 기능을 제공한다.

하지만 Print대화칸은 특수하다. Print대화칸은 View창문부품을 교체할 때까지 교체될수 없다. Qt는 QPrinter에 의하여 완전한 인쇄기구를 제공하지만 View창문부품을 교체할 때까지 그 리용을 시작할수 없다.

① Help대화칸의 삭제

Help대화칸도 역시 특수하다. Help대화칸을 사용자정의QDialog로 교체할 대신에 Qt방조체계클라스들을 리용하여 더 고급한 직결방조를 제공할수 있다. 그러나 Main Window 및 View창문부품들을 교체할 때까지 QToolTip, QWhatsThis와 같은 클라스의 리용을 시작할수 없다.

원시프로젝트들에 포함된 방조본문은 아주 적고 오직 XmNoteBook창문부품에 대한 정보만 포함한다. 도구암시와 whats-this본문의 사용은 이 실례에 충분하다. 자기의 이식프로젝트가 큰 직결방조체계를 가지고있다면 초본문항행과 완전인쇄기능을 가지는 풍부한 본문방조체계구축방법에 대한 정보를 《간단한 HTML방조열람기》실례를 통하여 혹은 Qt Assistant를 리용하여 볼것을 권고한다.

현존Help대화칸의 삭제는 상대적으로 간단하다. 프로젝트파일에서 Xmd/Help.h, Xmd/HelpP.h 그리고 Xmd/Help.c 파일들을 삭제하고 Makefile을 다시 생성한다.

Help차림표와 Help대화칸을 창조하는 코드는 todo.cpp에 있다. Xmd/Help.h머리부, *help_manager자원문자렬, help_cb()함수앞방향선언과 help_widget대역변수를 삭제한다. 또한 Help대화칸을 창조하는 코드를 삭제한다. 또한 Help차림표를 창조하는 코드를 삭제하여 차림표를 비운다. 끝으로 help_cb()함수실현을 삭제한다.

Help대화칸은 현재 프로젝트에서 삭제된다. 여전히 기본창문창문부품에 순간에 XmMainWindow을 사용하므로 QToolTip와 QWhatsThis방조를 추가할수 있다. 기

본창문과 올리펼침차림표계층을 이식하였다면 직결방조에 QToolTip와 QWhatsThis을 리용할수 있다.

3) Qt표준대화칸의 리용

Qt표준대화칸: QFileDialog와 QMessageBox의 리용을 시작한다. 두 클라스는 쓸모있는 정적편의함수들을 제공한다.

표 4-7. QFileDialog와 QMessageBox의 정적편의함수들

QFileDialog::getOpenFileName()	Open대화칸의 교체
QFileDialog::getSaveFileName()	Save As대화칸의 교체
QMessageBox::information()	Page Delete대화칸의 교체
QMessageBox::warning()	IO Error대화칸의 교체.

매개 함수는 QWidget *parent인수를 가진다. parent인수로서 0을 넘기면 기본창문의 중심에 배치되지 않고 화면의 중심에 배치된 대화칸들을 가진다. XmMainWindow를 창조하는 QMotifWidget를 가지고 이 클라스를 Motif대화칸와 Qt대화칸들에 부모로서 사용할수 있다.

todo.cpp에서 QMotifWidget와 QFileDialog용의 적당한 머리부를 포함하여야 한다.

다음으로 응용프로그람의 초기화방법을 약간 수정한다. Xt/Motif를 초기화하고 자체로 XtAppContext를 창조할수 있지만 QMotif는 이것을 할수 있다. 또한 QApplication가 X봉사기에로의 련결을 열게 한다. 다음에 xmMainWindowWidgetClass를 widgetclass인수로서 넘기여 QMotifWidget를 창조한다. 이제는 QMotifWidget::motifWidget()함수를 사용하여 Motif창문부품을 호출할수 있다. 쉘창문부품은 QMotifWidget에 의하여 자동적으로 창조된다. XtParent()를 리용하여 그것을 호출한다. 제일웃준위창문은 현재 QMotifWidget이며 이것은 Qt표준대화칸에 부모로서 리용할수 있다는것을 의미한다.

① Open과 Save As대화칸의 교체

우선 현존Motif파일선택대화칸의 모든 코드를 삭제한다. Xm/FileSB.h머리부, 대역변수file_dialog 그리고 대화칸을 창조하는 코드를 삭제한다. 또한 PresentFDialog()역호출함수를 삭제한다. 이 코드는 QFileDialog의 사용에 하나도 필요없다.

PresentFDialog()역호출함수를 삭제한 다음 Open과 Save As올리펼침차림표역호출이 Open()과 Save()함수들을 호출하게 만든다.

우선 두 함수의 선언을 변경해야 한다.

또한 인수들을 역호출로 변경한다. 제일웃준위QMotifWidget를 이 함수들에 client_data로 넘기므로 그것을 QFileDialog용 부모로 리용한다.

다음으로 Save()함수를 QFileDialog::getSaveFileName()를 사용하도록 수정한다.

그리고 Open()함수를 QFileDialog::getOpenFileName()를 리용하도록 수정한다.

프로젝트를 구축한 후에 응용프로그람을 실행하면 기대한대로 동작한다. Open과 Save As대화칸이 현재 QFileDialog를 사용한다.

② Page Delete와 IO Error대화칸의 교체

Page Delete대화칸이 창조되고 actions.c에서 사용된다. 이 파일을 C++로 이식할 필요가 있다. 이름을 actions.cpp로 변경하고 프로젝트파일을 수정하고 Makefile을 다시 생성한다.

actions.cpp를 콤파일하는데 필요한 변경은 적다. C머리부파일들과 대역변수들을 extern "C"블로크에 포함하여야 한다.

NewPage(), DeletePage(), EditPage() and SaveIt()함수들을 앞방향선언하여 콤파일러가 이 함수들의 정확한 기호들을 생성하게 한다.

하나의 무효지적자강제변환을 고착해야 한다.
그리고 Trim()함수에서 변수 new를 newstr 로 변경해야 한다.
이제는 DeletePage()함수를 QMessageBox::information()를 리용하도록 변경하여야 한다.
우선 QMessageBox의 적당한 머리부를 포함하는가 확인한다.
이 시점에서 코드에서 페지가 삭제되여야 한다. 이것을 수행하는 코드는 DoDeletePage()함수에 있다. DoDeletePage()의 내용을 이 점으로 이동하고 DoDeletePage()함수를 완전히 삭제한다.
다음에 todo.cpp를 변경하여 제일웃준위QMotifWidget를 client_data로서 DeletePage()함수에 넘긴다.
IO Error대화칸이 창조되고 io.c에서 리용된다. 이 파일을 C++로 이식해야 한다. 그것을 io.cpp로 변경하고 프로젝트파일을 수정하고 Makefile을 다시 생성한다.
io.cpp을 콤파일하게 하는데 필요한 변경은 극히 적다. C머리부파일들과 대역변수들을 extern "C" 블로크에 포함해야 한다.
ReadDB()와 SaveDB()함수들을 앞방향선언하여 콤파일러는 이 함수들에 정확한 기호들을 생성한다.
ParseNewLines()함수는 적당한 C++로 전환하여야 한다.
또한 PrintWithNewLines()함수는 적당한 C++로 변환해야 한다.
이제는 ReadDB()와 SaveDB()함수들을 변경하여 QMessageBox::warning()를 사용할수 있다.
우선 QMessageBox용 적당한 머리부를 포함하는가 확인한다.
프로젝트를 구축한 다음 응용프로그람을 실행하면 기대한대로 동작한다. 차이는 Page Delete와 IO Error대화칸이 현재 QMessageBox를 사용한다는것이다.
4) 사용자정의QDialog의 리용
표준대화칸을 교체한 후에 사용자정의대화칸들로 이동한다. 이 프로젝트는 단일사용자정의대화칸 Page Edit대화칸을 가진다.
코드를 자체로 작성하지 않고 Qt Designer로 대화칸을 설계한다. 사용자정의대화칸설계는 이 문서의 범위밖에 있다.
① Page Edit대화칸의 교체
Page Edit대화칸용 사용자정의QDialog서술은 pageeditdialog.ui에 보관된다. 다음 행을 추가하여 프로젝트파일.pro에 이 파일을 추가한다.
FORMS = pageeditdialog.ui
그리고 Makefile을 다시 생성한다. uic편의프로그람은 사용자정의QDialog코드를 생성하고 그다음 콤파일하여 응용프로그람에 련결한다. (uic는 .pro파일들로부터 생성된 makefile들로부터 자동적으로 호출된다.)
제일웃준위QMotifWidget를 EditPage함수에로의 client_data인수로 넘기여 새로운 PageEditDialog의 부모로 리용할수 있다. todo.cpp에서 Open과 Save As대화칸에서와 같은 방법으로 이것을 수행한다.
EditPage()함수는 actions.cpp에 실현된다. PageEditDialog와 QLineEdit에 필요한 머리부를 추가하는것으로 시작한다.
EditPage()함수에서 PageEditDialog를 창조하고 3개의 QLineEdit창문부품들의 초기값을 현재 페지로부터 들어온 값들로 설정하고 대화칸을 실행한다.
이 시점에서 코드에서 페지속성들은 수정되여야 한다. 이것을 수행하는 코드는 DoEditPage() 함수에 있다. DoEditPage()의 내용을 이 점으로 이동하고 DoEditPage()함수를 완전히 삭제한다.
page.h에 정의된 Page struct는 char* 배렬에 문자렬들을 보관한다.

PageEditDialog와 그것이 포함하는 자료가 이 함수로부터 돌아올 때 파괴되므로 국부부호화에서 유니코드 QString자료를 QCString으로 변환하고 그것을 duplicate it with qstrdup()를 가지고 복사할 필요가 있다.

갈은 과정을 minorTab본문에 대해서도 수행하고 majorTab본문에 대해서도 수행한다.

5) QMotifDialog에서 현존대화칸 리용

이미 언급한것처럼 Print대화칸은 View 창문부품으로 변환될 때까지 교체될수 없다. Print대화칸은 이식을 끝낸 다음에 QPrinter를 그대신 사용하므로 삭제된다. 이 정보에 기초하여 Print대화칸을 사용자정의QDialog로 교체할만한 가치가 없다고 생각한다. 그대신에 Motif기초대화칸을 유지하고 QMotifDialog를 리용하여 대화칸을 응용프로그람과 통합한다.

① 양식성요구

QDialog에서 양식성은 Motif와 다르다. QDialog::exec()함수는 대화칸이 끝날 때까지 돌아오지 않는다. Motif에서 이행금지성은 단지 쉘의 속성이고 프로그람작성자는 그 동작을 요구한다면 QDialog::exec()형식을 리용해야 한다.

QMotifDialog(QDialog의 파생클라스)를 사용하므로 QMotifDialog와 통합하는 각 대화칸에 대하여 accept와 reject역호출을 가져야 한다. 미리 정의된 Motif대화칸은 그것을 이미 가지고있다. 즉 XmNokCallback과 XmNcancelCallback역호출. 그러나 Print대화칸은 오직 accept역호출(XmdNprintCallback)만 가지고 reject역호출은 가지지 않는다. 이것을 추가해야 한다.

이것은 간단하다. Xmd/PrintP.h에서 XmNcancelCallback역호출에 대하여 XtCallbackList 를 추가한다.

Xmd/Print.c에서 XmdPrint 창문부품클라스용 자원목록에 XmNcancelCallback 역호출을 추가한다.

창문부품을 관리하지 않을 때는 이 역호출을 능동으로 해야 하며 do_help_cb()함수에서 그리고 unmanage_cb()함수에 있는 print 역호출을 능동으로 하지 말아야 한다.

② Print대화칸의 통합

그러면 Print대화칸이 적당한 accept와 reject역호출을 가지므로 QMotifDialog을 사용할수 있다. 우선 todo.cpp에 QMotifDialog머리부를 포함해야 한다.

인쇄대화칸을 창조하고 실행하는 ShowPrintDialog()함수를 추가한다.

Print차림표항목을 역호출로 변경하여 새로운 ShowPrintDialog()함수를 호출한다. 대화칸의 부모로 사용하는 제일 웃준위QMotifWidget를 넘긴다.

ShowPrintDialog()함수는 Print대화칸을 창조하고 실행한다. XmdNprintCallback역호출을 사용하여 대화칸을 받아들이고 XmNcancelCallback역호출을 사용하여 대화칸을 거부한다. QMotifDialog::acceptCallback()와 QMotifDialog::rejectCallback()함수를 각각 사용하여 이것을 간단히 수행할수 있다. 또한 print 역호출을 계속하여 이전처럼 Print()함수를 호출한다는것을 담보한다.

프로젝트를 구축한 후에 응용프로그람은 기대한대로 실행되고 동작한다. Print대화칸이 QMotifDialog를 리용하고있지만 시각적이거나 동작의 차이는 없다.

6) Qt기본창문클라스의 리용

대화칸을 모두 교체한 다음 기본창문을 바꿀 준비가 되였다. 우에서 언급한것처럼 현존XmMainWindow과 올리펼침차림표계층을 Qt기본창문클라스로 교체한다.

Qt Designer로 새 기본창문을 설계한다.

① 기본창문의 실현

기본창문에 대한 서술은 mainwindow.ui로서 보관된다. 프로젝트파일에 이 파일을 추가하고 Makefile을 다시 생성한다. uic편의프로그람은 기본창문의 코드를 생성한

다음 콤파일하여 응용프로그람에 련결된다.

또한 Qt Designer는 mainwindow.ui.h파일을 창조한다. 기본창문실현을 골격실현에 추가해야 한다.

QApplication와 QMotifWidget에 필요한 머리부들을 추가하는것으로 시작한다.

Motif역호출 struct들과 XmdPrint창문부품용 머리부를 포함한다.

이제는 기본창문안의 처리부들의 실현을 추가할 준비가 되였다. 차림표항목마다 하나의 처리부를 가진다. 매개 처리부는 todo.cpp와 actions.cpp에서 발견한 현존역호출을 호출한다.

표 4-8. 차림표항목과 역호출

File차림표		
New	MainWindow::fileNew()	New()역호출을 호출한다.
Open	MainWindow::fileOpen()	Open()역호출을 호출한다.
Save	MainWindow::fileSave()	SaveIt()역호출을 호출한다.
Save As	MainWindow::fileSaveAs()	Save()역호출을 호출한다.
Print	MainWindow::filePrint()	ShowPrintDialog()역호출을 호출한다.
Exit	MainWindow::fileExit()	QApplication::quit() 을 호출한다.
Selected차림표		
Properties	MainWindow::selProperties()	EditPage()역호출을 호출한다.
New	MainWindow::selNewPage()	NewPage()역호출을 호출한다.
Delete to Trash	MainWindow::selDeletePage()	DeletePage()역호출을 호출한다.

역호출을 사용하기전에 앞방향선언들을 추가한다.

현존역호출함수의 매개는 3개 인수를 가진다. 이 실례에서는 모든 인수에 NULL을넘긴다. 현존코드는 어떤 인수에도 의존하지 않는다. 매개의 처리부실현은 관련된 역호출함수를 호출하는 단일행이다. 코드는 그리 흥미없고 공간을 소비하므로 생략한다.

4가지 례외가 있다. Open(), Save(), EditPage() 그리고 DeletePage()역호출은 인수2(client_data인수)로서 제일웃준위QWidget 의 지적자를 넘긴다. 이 4개 함수들에서 둘째인수로서 QMainWindow에서 파생된 제일웃준위MainWindow인 this를 넘긴다.

② 기본창문의 교체

다음 단계는 응용프로그람에서 새로운 기본창문을 사용하는것이다. todo.cpp에서 필요한 변경은 대량의 코드를 삭제하였으므로 크다.

우선 새로운 기본창문용의 머리부를 추가한다.

Motif머리부의 대부분이 더는 필요없으므로 삭제한다.

QuitAppl()과 manageCB()역호출은 더는 필요하지 않으므로 그것들을 삭제한다. 대역shell변수도 필요없다. New(), Save() and Open()역호출에서 그것과 그 모든 참고를 삭제한다.

main()을 많이 변경한다. 우선 XmMainWindow을 가지는 QMotifWidget대신에 새로운 MainWindow을 사용한다.

이제는 QMotifWidget을 사용하여 XmNotebook 창문부품을 창조한다.

Motif차림표를 창조하는데 사용한 모든 코드를 삭제한다. main()의 나머지 코드는

자체로 해석하시오.

이제는 응용프로그람이 XmMainWindow대신에 QMainWindow을 사용한다. 프로젝트를 구축한 다음 응용프로그람은 기대한대로 실행되고 동작한다.

7) 현존코드의 재분해

다음에 서술한 단계들은 이식과정에 필요없지만 완전성에 필요하다.

① 자료구조를 C++에 이식

Page자료구조는 불투명한 자료형이다. 실제자료구조는 PageRec라고 부르며 Page는 PageRec의 지적자로 정의된다. 또한 PageRec구조체에 기억기를 할당하고 초기화하는 AllocPage()함수를 가지고있다.

C++에서는 구성자에서 이것을 수행한다. 또한 PageRec에서 모든 자원을 자동적으로 해방하는 구성자를 쓸수 있다.

PageRec구조체선언은 page.h에서 삭제된다. PageRec와 같은 자료성원, 구성자와 해체자를 가지는 Page구조체를 선언한다.

현존pages, currentPage 그리고 maxpages 대역변수들은 원천파일들로부터 삭제된다. page.h에서 그것들을 extern 선언으로 교체한다.

대역변수실례화는 todo.cpp에 선언된다.

매개 원천파일은 Page관련 대역변수들을 취급하는 함수선언들을 포함한다. 원천파일들로부터 이 선언들을 삭제하고 그것들을 page.h 머리부파일에서 선언한다.

Page가 구성자를 가지므로 AllocPage() 함수를 삭제한다. 그것은 더는 필요없다. AllocPage()에 대한 호출은 NewPage(), DeletePage() 그리고 ReadDB()함수들에서 new Page() 로 교체된다. 또한 delete pages[X]를 가지고 페지들을 해체하도록 코드를 교체한다. 여기서 X는 적당한 색인값이다. 이것은 ReadDB와 DeletePage()함수들에서 수행된다.

대역변수pages를 호출하는 코드는 Page구조체의 자료성원들이 변경되지 않았으므로 수정할 필요가 없다. 현존코드는 계속 동작한다.

page.h 에서 선언한 OptionsRec구조체도 갱신되며 우의 Page구조체와 같은 견본을 따른다.

또한 todo.cpp에 대역변수실례화도 있다.

대역변수 options를 호출하는 코드는 Options 구조체의 자료성원들을 변경하지 않았으므로 수정하지 않는다. 현존코드는 계속 동작한다.

② new와 delete의 사용

Page와 Options 구조체들의 해체자들은 XtFree()대신에 delete를 사용하여 모든 char* 성원들을 해체한다. 이것은 Xt/Motif로부터 이식하고있으므로 필요한 변경이다. Page구조체성원들을 new 와 delete(XtMalloc(), XtNewString() 그리고 XtFree() 대신에)를 사용하도록 수정하는 현존코드를 고착시켜야 한다.

todo.cpp에서 PageChange()함수는 단지 SetPage()를 호출하기 전에 현재 페지의 내용과 유표위치를 보관한다. Page구조체의 성원들을 수정할 때 new와 delete를 사용한다.

XmText창문부품의 내용을 보관할 때 qstrdup()를 리용하여 XmTextGetString()함수로부터 돌아온 문자렬의 사본을 만든다.

io.cpp에서 ReadDB()함수는 류사한 변경을 요구한다. XtMalloc()와 XtNewString()의 모든 사용을 new와 qstrdup()로 각각 교체한다.

이것은 바로 파일을 열기전에 새 페지를 시작할 때와 타브본문에서 읽어들일 때 수행해야 한다.

ReadDB()함수는 XtRealloc()를 리용하여 자료기억완충기를 전개한다. 불행하게도 C++는 현존자료블로크를 재할당하는 방법을 제공하지 않으므로 자체로 수행한다.

또한 ReadDB()에서 2의 인수로 XtMalloc()를 호출하는 경우가 한번 있다. 이것은 파일을 읽을수 없을 때 수행된다. 빈문자렬창조는 필요없으므로 이 코드를 삭제하고 그대신 new를 사용한다.

io.cpp에서 SaveDB()함수도 역시 이 변경을 요구한다. XtFree()의 한번의 출현을 delete로 변경한다.

끝으로 todo.cpp의 main()함수에서 XtNewString()의 두번의 출현을 교체한다.

③ 현존코드의 이동

재분배과정의 나머지는 현존코드를 새 위치로 이동하는것을 포함한다. 현재 mainwindow.ui.h파일의 각 함수는 단지 다른 파일들에 있는 낡은 역호출처리함수들을 호출한다. 낡은 역호출함수들을 호출할 대신에 실현이 그에 따라 이동된다.

표 4-9. 이동하는 함수들

함수	원시파일	이동하는 함수
New()	todo.cpp	MainWindow::fileNew()
Open()	todo.cpp	MainWindow::fileOpen()
SaveIt()	actions.cpp	MainWindow::fileSave()
Save()	todo.cpp	MainWindow::fileSaveAs()
ShowPrintDialog()	todo.cpp	MainWindow::filePrint()
EditPage()	actions.cpp	MainWindow::selProperties()
NewPage()	actions.cpp	MainWindow::selNewPage()
DeletePage()	actions.cpp	MainWindow::selDeletePage()

Print()역호출함수는 여전히 Print 대화칸으로 교체되므로 그것을 mainwindow.ui.h로 이동하고 static로 만든다.

이전에 Open(), Save(), EditPage() 그리고 DeletePage()함수들은 부모인수로서 client_data를 가지고 대화칸을 창조하였다. 코드를 직접 기본창문실현으로 이동하였으므로 부모인수로서 this를 리용하여 이 대화칸들을 창조한다.

PageChange()역호출함수는 actions.cpp로부터 todo.cpp로 옮기고 어디서나 사용하지 않으므로 static로 만들었다.

actions.cpp에 대한 초기의 실현은 Trim()함수가 여분으로 만들었으므로 그것을 삭제한다.

todo.cpp에서 MIN()과 MAX()마크로는 여분으로 된다. Qt는 우리가 사용할 QMIN()와 QMAX()마크로들을 제공한다.

초기의 수정이 fallback_resources 배렬을 여분으로 만들었으므로 삭제한다.

가까운 미래에 우리의 프로그람은 Motif를 더는 사용하지 않을것이며 더는 QMotif를 사용하지 않게 될것이다. 이것을 준비하기 위하여 resources과 optionDesc배렬을 삭제하고 APP_CLASS인수를 가지는 QMotif실례를 창조한다.

원천파일들에서 #include문들은 재분배변경으로 인하여 대체로 부정확하다. 대부분의 #include문들은 더는 필요하지 않다. 매개 파일로부터 #include 문들은 아래에 렬거하며 어느 머리부가 매개 파일에 삭제되고 추가되는가 서술하지 않는다.

actions.cpp용 머리부

io.cpp용 머리부

todo.cpp용 머리부

mainwindow.ui.h용 머리부

8) 보기창문부품의 교체

View창문부품을 교체할 준비가 되였다. 하지만 실례프로그람은 XmNotebook창문부품클라스를 사용한다. Qt는 이 클라스와 등가한것을 직접 제공하지 않으며 3가지 가능성이 있으며 매개는 우결함을 가진다.

① 현존Qt창문부품들을 사용하여 변환을 계속할수 있다.

우점 - Qt가 제공하는 창문부품들은 잘 설계되고 시험되였으며 사용자대면부를 고속으로 재설계할수 있다.

결함 - 대체로 현존자료구조와 코드를 수정하거나 다시 써야 한다. 새 코드는 응용프로그람의 이전판과의 호환성을 유지하는 방법으로 써야 한다.

② XmNotebook창문부품클라스와 등가한 QWidget의 새로운 파생클라스를 쓸수 있다.

우점 - 현존자료구조는 변경하지 않으며 이전과 앞으로의 판들과 호환성을 가지게 한다.

결함 - 새 창문부품을 쓰고 다시 시험해야 한다. 응용프로그람에서 현존코드는 새로운 창문부품의 API를 취급할수 있도록 변경되여야 한다.

③ XmNotebook 창문부품에 손을 대지 않게 할수 있다.

우점 - 현존자료구조와 코드는 변경되지 않으므로 다른 프로젝트들에 대하여 개발을 계속할수 있다.

결함 - 이것은 가장 단순한 해결이지만 응용프로그람은 여전히 X11에 의존하며 Qt가 유지하는 모든 가동환경에서 응용프로그람을 전개시킬수 없다.

첫째 수법으로 실례프로젝트의 이식을 완성하며 QTextEdit, QLabel 그리고 QSpinBox를 리용하여 류사한 형식을 제공한다. 유일한 차이는 타브를 가지고있지 않는것이다.

Qt Designer를 리용하여 QTextEdit, QLabel 및 QSpinBox창문부품을 기본창문창문부품에 추가한다.

① 자료구조수정

Page구조체는 삭제해야 할 majorPB와 minorPB성원들을 가지고있다.이 성원들은 현존판에서 현시된 타브들에 대응된다. 새 판은 타브를 가지지 않으므로 삭제한다.

② 코드수정

응용프로그람에서 현존함수들의 대부분은 새 View 창문부품과 작업하도록 수정되여야 한다. MainWindow클라스는 현존함수들에 대응하는 5개의 새 함수를 가진다.

표 4-10. 현존함수와 새 함수들

현존함수	새 함수
void SetPage(int)	void MainWindow::setPage(int)
void PageChange(...)	void MainWnidow::pageChange(int)
void TextChanged(...)	void MainWnidow::textChanged()
void ReadDB(char *)	void MainWindow::readDB(char *)
void SaveDB(char *)	void MainWindow::saveDB(char *)

SetPage()함수실현은 mainwindow.ui.h에서 MainWindow::setPage()함수로 옮긴다. page.h와 actions.cpp에서 각각 SetPage()함수선언과 실현을 삭제한다. MainWindow::setPage() 가 정확히 동작하게 하기 위하여 기본창문창문부품에서 새 창문부품을 사용하도록 코드를 수정한다.

우선 spinbox의 현재값을 현재페지번호로 설정한다.

다음에 textedit 의 현재본문과 유표위치를 현재페지의 내용으로 설정한다.

현재 페지가 사용자정의표식자를 가지면 그것을 textlabel의 현재본문으로 설정하고 그렇지 않으면 textlabel 내용을 "Page X"(여기서 X는 현재페지번호)로 설정한다.

현재페지가 크고작은 타브본문을 가지면 이것들을 labeltext에 추가한다. 이것은

사용자가 입력한 모든 정보가 보인다는것을 담보한다.

현재 페지가 존재하지 않을 가능성을 계속 조종해야 한다. 이 경우에 textedit창문부품의 내용을 삭제하고 textlabel 내용을 현재페지번호(페지가 무효라는 지적을 가진다)로 설정한다.

PageChange()함수는 todo.cpp으로부터 mainwindow.ui.h의 MainWindow::pageChange()함수로 옮긴다. MainWindow::setPae()함수에서처럼 코드가 기본창문창문부품의 새 창문부품들을 사용하도록 수정해야 한다.

알아두기: QTextEdit::text()는 표준char* 배렬로 변환할것을 요구하는 QString을 돌려준다. 그러자면 국부부호화에서 문자렬의 사본을 창조한다. QString::local8Bit()에 의하여 돌아온 QCString에 포함된 자료가 QCString이 해체될 때 해체되므로 qstrdup()을 리용하여 사본을 만들 필요가 있다.

TextChanged()함수는 modified 변수를 1로 설정하는것외에 다른것을 하지 않는다. 새 MainWindow::textChanged()함수는 정확히 같은 일을 한다.

MainWindow::pageChange()와 MainWindow::textChanged()함수들이 대역변수 modified 를 호출하므로 mainwindow.ui.h의 선두에 앞방향선언을 추가한다.

io.cpp에서 ReadDB()와 SaveDB()실현은 각각 MainWindow::readDB()과 MainWindow::saveDB()으로 이름을 변경한다. 코드가 적당히 작업하도록 수정하여야 한다.

우선 MainWindow, QSpinBox 및 QTextEdit클라스의 #include문을 추가한다.

새로운 MainWindow::readDB()와 MainWindow::saveDB()함수들은 Xt/Motif 함수를 사용하지 않으므로 Xt/Motif #include 문과 대역변수 notebook와 textw를 삭제한다. 이 함수들은 크게 변경되지 않고 이전판들과 호환을 유지한다. 또한 ReadDB()과 SaveDB() 함수들은 MainWindow 성원함수들로 변환되였으므로 this를 parent 인수로서 QMessageBox함수들에 넘길수 있다.

MainWindow::readDB()함수에서 파일을 읽어들인 후에 spinbox의 현재값과 최대값들을 적당히 설정한다.

MainWindow::saveDB()함수에서 현재 현시된 본문을 보관하므로 QTextEdit::text()를 XmTextGetString()대신에 리용한다.

알아두기: QTextEdit::text()은 표준char* 배렬로 변환할것을 요구하는 QString을 돌려준다. 그러자면 국부부호화에서 문자렬의 사본을 창조한다. QString::local8Bit()에 의하여 돌아온 QCString에 포함된 자료가 QCString이 해체될 때 해체되므로 qstrdup()을 리용하여 사본을 만들 필요가 있다.

Page구조체로부터 majorPB와 minorPB성원들의 삭제로 인하여 actions.cpp에서 FixPages()함수는 더는 필요없다. actions.cpp와 page.h로부터 FixPages()의 실현과 앞방향선언을 삭제한다. FixPages()호출은 MainWindow::selNewPage()와 MainWindow::selDeletePage()로부터 삭제되고 그것들은 둘다 mainwindow.ui.h에 있다.

AdjustPages()를 mainwindow.ui.h로 옮기고 이 파일에서만 리용하므로 static로 만든다. page.h로부터 앞방향선언을 삭제한다.

수정후에 actions.cpp 파일은 빈다. 프로젝트파일에서 그것을 삭제하고 Makefile을 다시 만든다.

새로운 View창문부품을 실현하였으므로 낡은 Motif 기초보기창문부품을 todo.cpp에서 삭제하여야 한다.

Motif창문부품을 하나도 사용하지 않으므로 qmotifwidget.h를 포함하는 모든 Motif #include 문을 삭제한다.

또한 ReadDB()함수의 앞방향선언과 notebook, textw 및 labelw 대역변수들을

삭제한다.

다음에 QMotifWidget를 사용하는 center창문부품을 삭제한다. 기본창문창문부품과 View 창문부품을 MainWindow 클라스에서 완전히 포함하므로 여분의 초기화는 mainwindow 창문부품창조후에 요구된다.

ReadDB()와 SetPage() 함수들이 MainWindow성원함수들로 변경되였으므로 mainwindow실례를 리용하여 그것들을 호출해야 한다.

View창문부품은 현재 교체되였다. 프로젝트를 구축하고 응용프로그람이 정확히 동작하는가 확인한다.

9) 인쇄대화칸의 교체

Print 대화칸은 응용프로그람에서 Motif를 사용하는 마지막 부분품이다. 현재 Print()함수는 일시파일에 평본문을 쓰는것외에 아무것도 하지 않고 'lpr'을 실행하여 본문을 인쇄기에 보낸다. QPrinter를 사용하므로 이 함수가 아무것도 하지 않으므로 삭제한다. 현재MainWindow::filePrint()실현은 삭제된다. 새로운 MainWindow::filePrint()실현을 mainwindow.ui.h에 쓴다.

① 인쇄에 리치본문의 리용

Qt는 HTML의 부분모임을 사용하는 리치본문을 제공한다. QSimpleRichText클라스는 리치본문인쇄를 간단하게 만든다. 우리가 해야 할 일은 적당한 위치에 삽입된 적당한 형식타그들을 창조하는것이다. 실례에서 인쇄출력을 이전판들과 비슷하게 유지한다.

우선 사용할 형식타그들을 창조한다.

다음에 모든 페지들을 순환하면서 페지표식자, 내용 및 형식문자들을 printtext변수(QString)에 추가한다.

MainWindow::filePrint()함수의 나머지는 실제의 인쇄코드이다. 여기서 간단히 우에서 창조한 문자렬을 리용하는 QSimpleRichText객체를 창조하고 이 문자렬을 QPainter를 리용하여 QPrinter객체에 그린다.

② Xt/Motif에서 의존관계의 삭제

응용프로그람은 Xt나 Motif창문부품들을 더는 사용하지 않는다. 이제는 Xt와 Motif에 대한 의존관계삭제를 완료할수 있다.

우선 #include 문들을 mainwindow.ui.h에서 삭제한다.

MainWindow::fileNew()함수는 Xt서고로부터 Boolean과 False 예약어들을 사용한다. C++는 이것들을 언어에 구축하였으므로 bool과 false를 대신에 사용한다.

응용프로그람에서 Xt와 Motif를 완전히 삭제하는데 필요한 마지막 수정은 QMotif클라스의 리용을 중지하는것이다. todo.cpp에서 qmotif.h #include문을 삭제하고 main()함수에서 실례화를 삭제한다.

이렇게 한 다음 프로젝트파일의 LIBS변수에서 -lXm와 -lqmotif를 삭제할수 있다. 프로젝트파일은 또한 응용프로그람에서 이전에 사용한 낡은 사용자정의Motif 창문부품들을 위한 원천과 머리부들이 포함되여있다. 이것들도 삭제한다.

Makefile을 다시 만들고 프로젝트를 구축한 다음 응용프로그람이 정확히 동작하는가 확인한다.

10) 개발의 계속

프로젝트에서 Motif를 더는 사용하지 않아도 Qt에로의 이식이 완전히 끝난것은 아니다. Qt는 즉시 리용할수 있는 수많은 유용한 특성을 제공한다. 가장 흥미있는것들중의 일부는 현존프로젝트의 확장을 시작하기 위한 차림표로서 아래에 제시한다.

① 유니코드리용

국제화지원은 Qt에서 아주 쉽다. 본문보관에서 char*대신에 QString의 리용은 세계의 대부분의 언어들에 대한 지원을 준다. Page와 Options구조체는 현재 더 간단해보인다.

Page와 Options구조체를 사용하는 모든 함수들은 QString을 적당히 사용하도록 갱신되여야 한다. 또한 QString은 암시적공유클라스이므로 문자렬들과 관련한 기억관리를 더는 하지 말아야 한다. qstrdup()함수의 출현을 모두 삭제하고 문자렬과 작업할 때 더는 new나 delete를 사용하지 말아야 한다. QString은 필요할 때 자료를 할당하고 삭제한다.

여기에 mainwindow.ui.h로부터 발취한 MainWindow::fileOpen()과 MainWindow:: pageChange()함수가 있다. 본문을 보관할 때 코드에서 delete 혹은 qstrdup() 를 더는 사용하지 않는다.

응용프로그람에서 함수들의 거의 대부분은 이 변경의 영향을 받는다. 대부분의 경우에 우리가 추가하는것보다 더 많은 코드의 삭제로 끝난다.

② 가동환경에 의존하지 않는 코드쓰기

Qt는 수많은 입출력클라스들을 제공한다. MainWindow::readDB()와 MainWindow::saveDB()에서 이것들을 사용한다. 현재 이 함수들은 UNIX콤퓨터들에서 발견한 함수들만 사용한다. QFile과 QTextStream사용은 UNIX에 대한 의존을 없애며 Microsoft Windows와 Apple Mac OS X에서 응용프로그람을 구축하고 시험할수 있다.

가동환경에 의존하지 않는 판의 MainWindow::readDB()와 MainWindow::saveDB()함수들은 io.cpp파일에 있다.

③ 현대사용자대면부의 설계

Qt Designer로 기본창문창문부품을 설계하였으므로 대면부를 간단히 확장할수 있다. 류동가능도구띠를 포함하는 QMainWindow의 고급한 특성들을 리용할수 있다. 이것은 Qt Designer 로 간단히 추가할수 있다. 프로젝트의 최종판은 Open, Save, Print, New Page 그리고 Delete to Trash 작용들의 고속호출을 제공하는 도구띠를 포함한다.

가능성은 끝이 없다. 공통의 Cut, Copy 및 Paste작용을 가지는 Edit차림표가 상대적으로 짧은 시간동안에 추가될수 있다. 우리의 프로젝트는 다른 가동환경으로 확장되며 보통탁상콤퓨터와 Qt/Embedded를 가지고 실행하는 수첩형장치사이에 todo목록을 동기화하게 하는 차림표와 대화칸들을 추가할수 있다.

제12절. Qt OpenGL확장에서 X11오버레이를 사용하는 방법

X11오버레이(overlay)는 화상을 파괴함이 없이 화상우에 주해를 그리기 위한 강력한 기구로서 대량의 화상묘사시간을 절약한다[1].

경고: 5.0판으로부터 Qt OpenGL확장은 OpenGL오버레이들의 사용을 위한 직접적인 지원을 포함한다. 오버레이의 수많은 사용에서 이것은 아래에 서술하는 기술이 여유를 가지게 한다(오버레이실례프로그람 참고). 다음은 오버레이평면들에서 비QGL창문부품들을 사용하는 방법에 대하여 설명한다.

전형적인 경우에 X11오버레이들은 현재판의 Qt와 Qt OpenGL확장과 함께 간단히 사용될수 있다. 다음의 요구를 적용한다.

① 자기의 X봉사기와 그라픽스카드/하드웨어는 오버레이를 지원해야 한다. 수많은 X봉사기들에서 오버레이지원은 환경구성선택에 의해 설정될수 있다(자기의 X봉사기설치문서 참고).

② 자기의 X봉사기는 오버레이시각(visual)을 기정시각으로 사용하도록 환경이 구성되여야 한다. 대다수 현대 X봉사기들은 이것을 수행하므로 올리펼침차림표, 겹침창문 등 추가적인 우점을 가지며 기본평면에서 기초화상들을 파괴하지 않으므로 품이 드는 재

그리기를 피한다.

③ OpenGL묘사를 위한 가장 좋은(가장 깊은) 시각은 기본평면에 있다. 이것은 보통의 경우이다. 전형적으로 오버레이를 유지하는 X봉사기들은 기본평면에서 24bit깊이 TrueColor시각을 제공하고 오버레이평면에서 8 bit PseudoColor(기정)시각을 제공한다. 제공된 실례프로그람 X11오버레이는 이것들을 검사하고 오유를 알린다.

1. 작업방법

우에 주어진것처럼 QGLWidget는 기정으로 기본평면시각을 리용하는 한편 다른 모든 창문부품들은 오버레이시각을 리용한다. 이리하여 QGLWidget의 꽉대기우에 표준창문부품을 배치하고 OpenGL창문안의 화상을 파괴하지 않고 표준창문부품우에 그리기할수 있다. 다시 말하면 QPainter의 그리기능력을 모두 발휘하여 주해, 선택창(rubberband) 등을 그릴수 있다. 이것은 오버레이의 전형적인 사용으로서 주해를 그리는데 OpenGL을 사용하기보다 훨씬 더 쉽다.

오버레이평면은 투명색이라는 고유한 색을 가진다. 이 색으로 그린 화소들은 보이지 않으며 대신에 기초에 놓여있는 OpenGL화상은 들여다보인다. 실례프로그람 X11오버레이에서 파일main.cpp은 투명색을 포함하는 QColor을 돌려주는 루틴을 포함한다. 오버레이창문부품에서는 일반적으로 배경색을 투명색으로 설정하려고 하므로 OpenGL화상은 그것이 명백히 겹쳐그려지는 경우를 제외하고 들여다보인다.

알아두기: 이 기술을 사용하려면 QApplication에 "ManyColor" 혹은 "TrueColor" ColorSpec를 사용하지 말아야 한다. 왜냐하면 필요할 때 오버레이평면이 아니라 보통 기본평면에 있는 표준Qt창문부품들이 TrueColor시각을 사용하기때문이다.

2. X11시각에 대하여

utilities등록부에는 자기 X봉사기의 능력을 결정할수 있도록 방조할수 있는 두개의 작은 프로그람들이 포함되여있다.

glxvisuals는 X봉사기가 제공하는 모든 GL능력이 있는 시각들을 매개에 대한 깊이와 기타 GL고유정보와 함께 렬거한다. 특히 "lvl"란을 지적한다. 이 란안의 수자는 시각이 오버레이평면내에 있다는것을 의미한다.

sovinfo는 쓸수 있는 모든 시각들을 렬거하며 오버레이시각들의 특별한 투명정보를 제공한다.

X11오버레이실례프로그람은 표준Qt창문부품들에 어떤 시각이 사용되는가와 QGLWidget에 의해 어떤 시각이 사용되는가를 출력한다.

제13절. 응용프로그람그림기호의 설정

일반적으로 응용프로그람의 왼쪽웃구석에 표시되는 응용프로그람그림기호는 제일웃준위창문부품에 대하여 QWidget::setIcon()호출에 의하여 설정된다.

실행가능응용프로그람파일자체의 그림기호를 변경하기 위하여서는 탁상에 그것이 표시될 때(즉 응용프로그람의 실행에 앞서) 가동환경에 의존하는 기술을 받아들여야 한다.

1. Windows에서 응용프로그람그림기호의 설정

우선 그림기호화상을 포함하는 ICO형식비트매프파일을 창조한다. 이것은 Microsoft Visual C++에서 File|New...를 선택하고 그때 나타나는 대화칸의 File타브를 선택하고 Icon을 누르는 방법으로 수행할수 있다. (자기의 응용프로그람을 Visual C++에 적재할 필요는 없으며 오직 그림기호편집기만 리용한다.)

자기 응용프로그람의 원천코드등록부에 ICO파일 례를 들면 myappico.ico라는 이

름으로 보관한다. 그다음 다음과 같이 한행의 본문을 넣은 본문파일 실례로 myapp.rc를 작성한다.

IDI_ICON1ICON DISCARDABLE "myappico.ico"

끝으로 makefile생성에 qmake를 사용하는것을 전제로 하여 다음의 행을 자기의 myapp.pro파일에 추가한다.

RC_FILE = myapp.rc

makefile과 응용프로그람을 다시 생성한다. 이제는 .exe파일이 Explorer에서 그림기호로 표시된다.

qmake를 사용하지 않을 때 필요한 단계들은 우선 .rc파일에서 rc프로그람을 실행하고 응용프로그람을 생성된 .res파일과 련결한다.

2. Mac OS X에서 응용프로그람그림기호의 설정

일반적으로 응용프로그람의 류동(dock)령역에 표시되는 응용프로그람그림기호는 제일 웃준위의 창문부품에서 QWidget::setIcon()호출에 의하여 설정된다. 이것은 프로그람이 함수호출전에 응용프로그람의 류동령역에 표시할수 있게 하며 이 경우에 기정그림기호는 동화상이 동작할 때 나타난다.

정확한 그림기호이 나타난다는것을 담보하기 위하여 응용프로그람이 기동되고있을 때와 Finder에서 가동환경에 의존하는 기술을 받아들이는것이 필요하다.

대부분의 프로그람들이 그림기호파일(.icns)을 창조할수 있지만 권고하는 수법은 Apple에 의하여 제공되는 *Icon Composer*프로그람(Developer/Application홀더에서)을 리용하는것이다. *Icon Composer*는 몇가지 각이한 크기의 그림기호를 반입하는것은 물론 각이한 상황에서 리용할수 있게 한다. 프로젝트등록부에 그림기호들의 모임을 하나의 파일로 보관한다.

makefile생성에 qmake를 사용한다면 .pro프로젝트파일에 다음의 한행을 추가해야 한다. 실례로 그림기호파일의 이름이 myapp.icns이고 프로젝트파일이 myapp.pro이면 다음 행을 myapp.pro에 추가한다.

RC_FILE = myapp.icns

이것은 qmake가 적당한 위치에 그림기호들을 넣고 그림기호를 위한 Info.plist항목을 창조한다는것을 담보한다.

qmake를 사용하지 않는다면 다음과 같이 수동적으로 수행한다.

① Developer/Applications에 있는 PropertyListEditor를 사용하여 응용프로그람용의 Info.plist파일을 창조한다.

② 다시 PropertyListEditor를 리용하여 .icns레코드를 Info.plist파일안의 CFBundleIconFile레코드와 련결한다.

③ 자기의 응용프로그람묶음Resource등록부에 icns과 Info.plist를 모두 복사한다.

3. 일반Linux탁상에서 응용프로그람그림기호의 설정

여기서는 두가지 일반Linux탁상환경 KDE와 GNOME에서 응용프로그람의 그림기호를 제공하는 방법을 간단히 서술한다. 응용프로그람그림기호들을 서술하는데 사용하는 핵심기술은 두 탁상에서 같으며 다른데에도 적용할수 있지만 매개에는 고유한 특성이 있다.

흔히 사용자들은 실행가능파일을 직접 사용하지 않지만 그대신에 응용프로그람을 탁상우의 그림기호를 선택하여 기동한다. 이 그림기호들은 그 그림기호에 대한 정보를 비롯한 응용프로그람의 서술을 포함하는 탁상항목파일들의 표시이다. 두 탁상환경은 이 파일들의 정보를 얻을수 있으며 그것을 리용하여 탁상우에, 시작차림표에 그리고 조종판에 응용프로그람에 대한 지름건를 생성한다.

탁상항목파일들을 응용프로그람의 세부를 효과적으로 밀봉할수 있지만 여전히 각

탁상환경에 편리한 위치에 그림기호들을 보관해야 한다.

비록 그림기호들을 찾는데 리용되는 경로가 사용하는 탁상과 그 환경구성, 등록부구조에 의존하지만 그 매개는 같은 패턴을 따라야 하며 보조등록부들은 주제(theme), 그림기호크기, 응용프로그람형에 따라서 배렬된다. 일반적으로 응용프로그람그림기호들은 hicolor주제에 추가되므로 크기가 32화소인 바른4각형응용프로그람그림기호는 그림기호경로아래의 hicolor/32x32/apps등록부에 보관되군한다.

① KDE

응용프로그람그림기호들은 모든 사용자들이 리용하거나 개별적인 사용자가 사용하기 위하여 설치할수 있다. 현재 KDE탁상에 가입한 사용자는 kde-config에 의하여 실례로 말단창문에서 다음과 같이 입력하여 그 위치를 발견할수 있다.

kde-config --path icon

일반적으로 stdout에 출력된 두점으로 구분된 경로목록은 사용자에게 고유한 그림기호경로와 체계범위의 경로를 포함한다. 이 등록부들아래에서 그림기호주제사양에 서술된 관례에 따라 그림기호들을 찾고 설치할수 있어야 한다.

KDE전용으로 개발하고있다면 KDE구축체계의 우점을 리용하여 응용프로그람의 환경을 구성해야 한다. 이것은 그림기호들이 KDE용의 적당한 위치에 설치되도록 담보한다.

② GNOME

응용프로그람그림기호들을 구성방식에 의존하지 않는 파일들을 포함하는 표준체계범위등록부안에 보관된다. 이 위치는 gnome-config에 의해 결정할수 있으며 실례로 말단창문에서 다음과 같이 입력할수 있다.

gnome-config --datadir

stdout에 출력된 경로는 pixmaps라는 등록부를 포함하는 위치를 참고하고 pixmaps등록부안의 등록부구조는 그림기호주제사양에 서술된다.

GNOME전용으로 개발한다면 GNU구축도구의 표준일식을 사용하려고 할수 있다. (이것도 GTK+/Gnome응용프로그람개발서적의 적절한 절에서 서술한다.) 이것은 그림기호들이 GNOME용 적당한 위치에 설치되도록 담보한다.

제14절. 쎄숀관리

1. 정의

쎄숀(session)은 각각 특정한 상태를 가지는 실행중에 있는 응용프로그람들의 그룹이다. 쎄숀은 쎄숀관리기라는 기구에 의해 조종된다. 쎄숀에 참가하는 응용프로그람들을 쎄숀의뢰기라고 부른다.

쎄숀관리기는 의뢰기에 사용자를 위한 지령을 내보낸다. 이 지령들은 의뢰기들이 보관하지 않은 변경의 보관(실례로 열린 파일들을 보관하여)을 맡기거나 앞으로의 쎄숀들을 위하여 그것들의 상태를 보존하거나 단순히 완료한다. 이 조작들의 모임을 쎄숀관리라고 한다.

일반적인 경우에 쎄숀은 사용자가 탁상에서 한번에 실행하는 모든 응용프로그람들로 이루어진다. 그러나 Unix/X11하에서 쎄숀은 각이한 콤퓨터들에서 실행하고있는 응용프로그람들을 포함할수 있으며 여러개의 현시기를 사용할수 있다.

2. 쎄숀의 완료

쎄숀은 보통 사용자가 탈퇴하려고 할 때 쎄숀관리기에 의해 완료된다. 또한 체계는 비상상태에서 실례로 전원이 꺼진 경우에 자동중지를 수행할수 있다. 틀림없이 이런 형의 중지들사이에 중요한 차이가 있다. 처음에 사용자는 응용프로그람과 교제하여 어느 파일들을 보관하고 어느 파일들을 무시하겠는가를 명백히 지정할수 있다. 파일들을 무시

하는 경우에는 교제할 필요가 없다. 콤퓨터앞에 앉아있는 사용자가 한명도 없을수 있다.

3. 통신규약과 각이한 가동환경에 대한 지원

Mac OS X와 MS-Windows에는 아직 응용프로그람들을 위한 완전한 쎄숀관리가 아직 없으며 이전 쎄숀들의 복귀도 없다. 이 조작체계들은 응용프로그람들이 사용자로부터 확인을 받은 후에 프로쎄스를 취소할 기회를 가지는 단순한 탈퇴(logout)를 지원한다. 이것은 QApplication::commitData()메쏘드에 대응하는 기능이다.

X11은 X11R6부터 완전한 쎄숀관리를 유지하였다.

4. Qt에 의한 쎄숀관리

QApplication::commitData()를 재정의함으로써 자기 응용프로그람이 고상한 탈퇴처리를 하게 하는것으로 시작하자. 오직 MS-Windows가동환경만 목표하고있다면 이것은 가능한껏 제공되여야 한다. 실제로 자기의 응용프로그람은 다음과 류사한 중지대화칸을 제공해야 한다.(이 대화칸에 대한 코드는 QSessionManager::allowsInteraction()의 문서에 있다.)

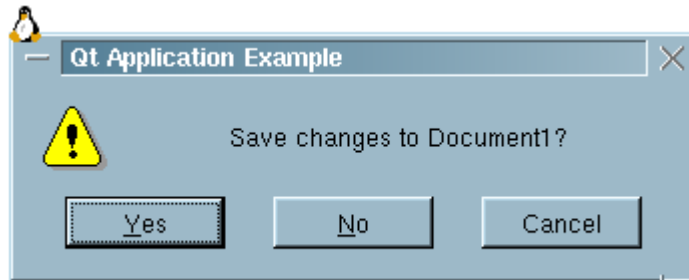



그림 4-2. 중지대화칸

또한 완전한 쎄숀관리(현재 X11R6에서만 지원됨)를 위해서는 응용프로그람의 상태보관과 잠재적으로 쎄숀의 다음 생명주기에서 상태의 되살리기를 고려해야 한다. 이 보관은 QApplication::saveState()의 재정의에 의해 수행된다. 이 함수에서 보관하고있는 모든 상태자료는 쎄숀식별자 QApplication::sessionId()에 의해 표식되여야 한다. 응용프로그람에 고유한 식별자는 대역적으로 유일하므로 충돌이 생기지 않는다(특정한 Qt응용프로그람의 상태보관과 되살리기에 대한 정보는 QSessionManager를 참고).

되살리기는 보통 응용프로그람의 main()함수에서 수행된다. QApplication::isSessionRestored()가 TRUE인가 검사한다. 그 경우에 다시 쎄숀식별자 QApplication:: sessionId()를 사용하여 자기의 상태자료를 호출하고 응용프로그람의 상태를 되살린다.

중요: 창문관리기가 탄창보관순서 혹은 기하정보와 같은 창문속성들을 되살리기 위해서는 자기의 유일한 응용프로그람범위의 객체이름들을 가지는 제일웃준위의 창문부품들을 식별하여야 한다(QObject::setName()참고). 응용프로그람을 되살릴 때 되살아나는 제일 웃준위의 모든 창문부품들이 이전에 가지고있던것과 같은 유일한 이름들을 가지도록 담보하여야 한다.

5. 쎄숀관리의 시험과 오유수정

Mac OS X와 Windows에서 쎄숀관리지원은 조작체계 그 자체에 이런 기능이 없는것으로 하여 상당히 제한된다. 단순히 쎄숀을 중지하고 자기의 응용프로그람이 기대한대로 동작하는가를 확인한다. 자기 응용프로그람을 기동하기전에 다른 응용프로그람, 보통 통합개발환경을 기동하는데 사용할수 있다. 이러한 다른 응용프로그람은 후에 중지통보를 얻으므로 중지를 취소하게 한다. 그렇지 않으면 매번 시험후에 다시 가입(log in)

해야 하는데 이것은 그자체에 문제가 없지만 시간을 소비한다.

Unix에서는 표준X11R6쎄숀관리를 유지하는 탁상환경을 사용하거나 X협회에 의해 제공된 쎄숀관리기참고실현을 사용할수 있다. 이 본보기관리기를 xsm이라고 부르며 표준X11R6설치의 부분이다. X11에서는 항상 사용할수 있는 man페지가 제공된다. xsm의 사용은 간단하다. (Athena기초사용자대면부와 다르다.) 여기에 간단한 수법이 있다.

- X11R6을 실행한다.
- 단일행을 포함하는 자기의 홈등록부에 점파일 .xsmstartup을 창조한다.

```
xterm
```

이것은 xsm에게 기정 및 실패에 안전한 쎄숀이 바로 xterm이라는것을 말한다. 그렇지 않으면 xsm은 그리 도움이 되지 않는 창문관리기 twm을 포함하는 많은 의뢰기들을 호출하려고 한다.

- 이제 다른 말단창문으로부터 xsm을 호출한다. 쎄숀관리기창문과 xterm이 둘다 나타난다. xterm은 현재 실행하고있는 다른 모든 쉘로부터 분리하여 설정되는 좋은 속성을 가지고있다. 이 쉘안에서 SESSION_MANAGER환경변수는 바로 자기가 기동한 쎄숀관리기를 지적한다.
- 새로운 xterm창문으로부터 자기의 응용프로그람을 호출한다. 그자체를 쎄숀관리기에 자동적으로 련결한다. 련결이 성공하였는가는 ClientList누름단추를 리용하여 검사할수 있다.

알아두기: 쎄숀관리되는 의뢰기들을 기동하거나 완료할 때 ClientList는 절대로 열려있지 않다. 그렇지 않으면 xsm은 중단될수 있다.

- 쎄숀관리기의 Checkpoint와 Shutdown단추들에 서로 다른 설정값들을 사용하여 자기 응용프로그람의 동작을 고찰해본다. 보관형 local은 의뢰기들이 그 상태를 보관해야 한다는것을 의미한다. 이것은 QApplication::saveState()함수에 대응된다. global보관형은 응용프로그람들에게 보관되지 않은 변경을 영원히 대역호출기억기에 보관할것을 요구한다. 이것은 QApplication::commitData()를 호출한다.
- xsm은 사용자의 탁상에서 사용가능한 쎄숀관리기의 존재로부터 분리되여있다. 시험환경으로 봉사하는데 충분히 안정하고 쓸모있다.

제15절. 공유클라스

Qt 의 수많은 C++클라스들은 명시적(explicit)이고 암시적(implicit)인 자료공유를 사용하여 자원리용률을 최대화하고 자료복사를 최소화한다.

공유클라스는 참고계수와 자료를 포함하는 공유자료블로크의 지적자로 이루어진다.

공유객체가 창조될 때 참고계수는 1로 설정된다. 참고계수는 새 객체가 공유자료를 참고할 때마다 늘어나고 객체가 공유자료참고를 해제할 때마다 줄어든다. 참고계수가 0으로 될 때 공유자료는 삭제된다.

공유객체들을 취급할 때 객체를 복사하는 두가지 방법 즉 깊은 복사와 얕은 복사가 있다. 깊은 복사(deep copy)는 객체의 복제를 암시한다. 얕은 복사(shallow copy)는 참고복사 즉 공유자료블로크의 지적자이다. 깊은 복사는 기억기와 CPU의 측면에서 비용이 든다. 얕은 복사는 아주 고속이다. 그것은 지적자의 설정과 참고계수의 증가만 포함하기때문이다.

명시적 및 암시적공유객체들을 위한 객체대입(operator=())은 얕은 복사에 의하여 실현된다. 깊은 복사는 copy()함수의 호출이나 QDeepCopy를 리용하여 이루어진다.

공유의 리득은 필요없이 자료를 복제할 필요가 없는것이고 그 결과 작은 기억기를 사용하고 자료의 복사가 적어진다. 객체들을 간단히 대입하고 함수인수로서 보내고 함수로부터 돌려줄수 있다.

그러면 명시적 및 암시적공유사이의 차이를 보자. 명시적공유는 프로그람작성자가 객체들이 공통자료를 공유한다는 사실을 알아야 한다는것을 의미한다. 암시적공유는 공유기구가 배경에서 발생하므로 프로그람작성자가 그에 대해 걱정할 필요가 없다는것을 의미한다.

1. QByteArray 실례

QByteArray는 명시적공유를 사용하는 공유클라스의 실례이다. 실례:

```
//                        Line    a=         b=         c=
QByteArray a(3),b(2)      // 1:   {?,?,?}   {?,?}
b[0] = 12; b[1] = 34;     // 2:   {?,?,?}   {12,34}
a = b;                    // 3:   {12,34}   {12,34}
a[1] = 56;                // 4:   {12,56}   {12,56}
QByteArray c = a;         // 5:   {12,56}   {12,56}   {12,56}
a.detach();               // 6:   {12,56}   {12,56}   {12,56}
a[1] = 78;                // 7:   {12,78}   {12,56}   {12,56}
b = a.copy();             // 8:   {12,78}   {12,78}   {12,56}
a[1] = 90;                // 9:   {12,90}   {12,78}   {12,56}
```

3행의 대입 a = b는 a의 원시공유블로크를 버리고(참고계수는 0으로 된다) a의 공유블로크가 b의 공유블로크를 지적하게 설정하고 참고계수를 증가시킨다.

4행에서 a의 내용은 수정되고 a 와 b 가 같은 자료블로크를 참고하므로 b도 역시 수정된다. 이것은 명시적공유와 암시적공유사이의 차이이다.

6행에서 객체 a는 공통자료로부터 분리된다. 분리는 객체가 그 자체의 비공개자료를 가진다는것을 확인하도록 공유자료가 복사된다는것을 의미한다. 그러므로 7행에서 a의 수정은 b나 c에 영향을 주지 않는다.

끝으로 8행에서는 a의 깊은 복사를 만들고 그것을 b에 대입하여 9행에서 a가 수정될 때 b는 변경되지 않는다.

2. 명시적공유와 암시적공유

암시적공유는 객체가 변경되려고 하고 참고계수가 1보다 크면 자동적으로 그 객체를 공유블로크에서 분리한다. (이것을 흔히 《써넣을 때의 복사(copy-on-write)》라고 한다.) 명시적공유는 이 일감을 프로그람작성자에게 넘긴다. 명시적공유객체가 분리되지 않으면 객체의 변경은 같은 자료를 참고하는 다른 모든 객체들을 변경하게 된다.

암시적공유는 이러한 부작용이 없이 기억기의 사용과 자료의 복사를 최적화한다. 그러면 왜 모든 공유클라스들에 암시적공유를 실현하지 않았는가? 대답은 QByteArray와 같은 그 내부자료에 대한 직접호출을 허용하는 클라스를 효과성때문에 명시적으로 공유할수 없다는것이다. 그것은 QByteArray가 모르게 변경될수 있기때문이다.

암시적공유클라스는 그 내부자료의 종합적조종을 가진다. 따라서 그 자료를 수정하는 임의의 성원함수들에서 자료를 수정하기전에 자동적으로 분리한다.

암시적공유를 사용하는 QPen클라스는 내부자료를 변경하는 모든 성원함수들에서 공유자료로부터 분리한다.

코드부분:

```
void QPen::setStyle( PenStyle s )
{
  detach();      // detach from common data
  data->style = s; // set the style member
}

void QPen::detach()
```

```
{
  if ( data->count != 1 ) // only if >1 reference
    *this = copy();
}
```

이것은 프로그람작성자가 다음과 같이 할수 있으므로 QByteArray에는 명백히 불가능하다.

```
QByteArray array( 10 );
array.fill( 'a' );
array[0] = 'f';    // will modify array
array.data()[1] = 'i'; // will modify array
```

QByteArray에서의 변화를 감시한다면 QByteArray클라스는 접수할수 없을 정도로 떠진다.

3. 명시적공유클라스

QMemArray형판클라스의 실레인 모든 클라스들은 명시적으로 공유된다.

QBitArray, QPointArray, QByteArray, QMemArray<type>의 다른 실례작성이 클라스들은 자기의 객체가 공유자료의 비공개사본을 얻으려고 하면 호출될수 있는 detach()함수를 가지며 참고계수 1을 가지는 깊은 사본을 돌려주는 copy()함수를 가진다.

이러한것은 QMemArray를 계승하지 않는 QImage에서도 같다. QMovie도 역시 암시적으로 공유되지만 detach()나 copy()를 가지지 않는다.

4. 암시적공유클라스

암시적으로 공유하는 Qt클라스들은 다음과 같다.

QBitmap, QBrush, QCursor, QFont, QFontInfo, QFontMetrics, QIconSet, QMap,QPalette, QPen, QPicture, QPixmap, QRegion, QRegExp, QString, QStringList,QValueList, QValueStack

이 클라스들은 객체를 변경하려고 할 때 자동적으로 공통자료로부터 객체를 분리한다. 프로그람작성자는 객체들이 공유된다는것을 전혀 알리지 않는다. 이리하여 클라스들의 개별적인 실례들을 개별적객체로 취급해야 한다. 그것들은 늘 개별적객체들로 동작하지만 가능할 때마다 추가적인 공유객체의 우점을 가진다. 이러한 리유로 복사비에 관계없이 함수들에 이 클라스들의 실례들을 값에 의해 인수로서 넘길수 있다.

실례:

```
QPixmap p1, p2;
p1.load( "image.bmp" );
p2 = p1; // p1과 p2은 자료를 공유한다
QPainter paint;
paint.begin( &p2 );     // cuts p2 loose from p1
paint.drawText( 0,50, "Hi" );
paint.end();
```

이 실례에서 p2에 대하여 QPainter::begin()가 호출될 때까지 p1와 p2은 자료를 공유한다. 왜냐하면 픽스매프의 그리기가 그 자료를 수정하지 않기때문이다. 또한 어떤것이 p2로 bitBlt()되지 않으면 같은 일이 발생한다.

경고: 자기가 암시적공유용기(QMap, QValueVector, 등)에 관심을 가지지 않을 때 그것을 복사하지 말아야 한다.

5. QCString은 암시적인가 명시적인가?

QCString은 암시적공유와 명시적공유를 혼합하여 사용한다. QByteArray로부터 계승된 data()와 같은 함수들은 명시적공유를 받아들이고 QCString안의 함수들은 자동

분리한다. 이처럼 QCString은 주로 Qt 1.x로부터 Qt 2.0으로 간단히 이식하기 위하여 제공되는 전문가전용클라스이다. 순수한 명시적공유클라스인 QString를 사용할것을 권고한다.

제16절. 형식개괄

Qt에서 형식(style)은 특별한 가동환경용의 GUI에서 찾게 되는 형식을 실현한다. 실례로 Windows가동환경은 Windows 혹은 Windows XP형식을 사용할수 있고 Unix가동환경은 Motif형식을 리용할수 있다.

이것은 Qt 3.x형식API를 가지고 사용자정의형식을 창조하고 리용하도록 하는데 요구되는 단계들을 보여주는 간단한 차림표이다. 우선 필요한 단계들을 거쳐서 형식을 생성한다. 즉

① 계승하려는 기초형식을 선택한다.

② 파생클라스에서 필요한 함수들을 재정의한다.

그다음 자기 응용프로그람이나 현존 Qt응용프로그람들에서 사용할수 있는 플라그인으로서 새 형식을 사용하는 방법을 설명한다.

- 사용자정의형식창조

① 계승하려는 기초형식을 선택한다.

첫 단계는 Qt에 제공된 기본형식들중 하나를 선택하여 자기의 사용자정의형식을 구축하는것이다. 그 선택은 자기가 얻으려는 형식에 의존한다. QWindowsStyle파생클라스 혹은 QMotifStyle파생클라스로부터 선택할것을 권고한다. 이 클라스들은 Qt형식엔진의 두가지 기초형식클라스들이다. QCommonStyle로부터의 직접계승은 자기 형식을 새로 실현하려고 하는 경우의 선택이다. 이 간단한 실례에서는 QWindowsStyle로부터 계승한다.

② 파생클라스에서 필요한 함수들을 재정의한다.

기초형식의 어느 부분을 변경하려고 하는가에 따라서 이 부분의 대면부를 그리는데 사용되는 함수들을 재정의하여야 한다. QStyle문서를 조사해보면 각이한 통지(primitive)와 조종, 복합조종들의 목록을 볼수 있다. 이 실례에서는 우선 QWindowsStyle에서 쓰이는 표준화살표들의 모양을 변경한다. 화살표들은 drawPrimitive()함수에 의해 그려지는 PrimitiveElement이므로 그 함수를 재정의하여야 한다. 다음의 클라스선언이 요구된다.

```
#include <qwindowsstyle.h>

class CustomStyle : public QWindowsStyle {
  Q_OBJECT
public:
  CustomStyle();
  ~CustomStyle();

  void drawPrimitive( PrimitiveElement pe, QPainter *p, const QRect & r,
   const QColorGroup & cg, SFlags flags = Style_Default,
   const QStyleOption & = QStyleOption::Default ) const;

private:
  // 복사구성자와 연산자=를 금지한다
```

```
  CustomStyle( const CustomStyle & );
  CustomStyle& operator=( const CustomStyle & );
};
```

우리의 형식에서 복사구성자와 =연산자를 허용하지 않는다. QObject는 Qt의 모든 형식클라스들의 기초클라스이며 QObject는 복사할수 없다는 자기의 견해가 있으므로 복사될수 없다.

QStyle문서로부터 PE_ArrowUp, PE_ArrowDown, PE_ArrowLeft 그리고 PE_ArrowRight가 무엇인가 수행하는데 요구되는 통지라는것을 알수 있다. drawPrimitive()함수에서 다음과 같은것을 얻게 된다.

```
CustomStyle::CustomStyle()
{
}

CustomStyle::~CustomStyle()
{
}
void CustomStyle::drawPrimitive( PrimitiveElement pe, QPainter * p, const QRect & r,
        const QColorGroup & cg, SFlags flags, const QStyleOption & opt ) const
{
  // 화살표에만 관심을 가진다
  if (pe >= PE_ArrowUp && pe <= PE_ArrowLeft) {
    QPointArray pa( 3 );
    // 화살표가 그리려는 구역 절반을 포함하게 한다
    int x = r.x();
    int y = r.y();
    int w = r.width() / 2;
    int h = r.height() / 2;
    x += (r.width() - w) / 2;
    y += (r.height() - h) /2;

    switch( pe ) {
    case PE_ArrowDown:
      pa.setPoint( 0, x, y );
      pa.setPoint( 1, x + w, y );
      pa.setPoint( 2, x + w / 2, y + h );
      break;
    case PE_ArrowUp:
      pa.setPoint( 0, x, y + h );
      pa.setPoint( 1, x + w, y + h );
      pa.setPoint( 2, x + w / 2, y );
      break;
    case PE_ArrowLeft:
      pa.setPoint( 0, x + w, y );
      pa.setPoint( 1, x + w, y + h );
```

```
      pa.setPoint( 2, x, y + h / 2 );
      break;
    case PE_ArrowRight:
      pa.setPoint( 0, x, y );
      pa.setPoint( 1, x, y + h );
      pa.setPoint( 2, x + w, y + h / 2 );
      break;
    default: break;

    }

    // 다른 색을 리용하여 화살표가 허용/금지된다는것을 가리킨다
    if ( flags & Style_Enabled ) {
      p->setPen( cg.mid() );
      p->setBrush( cg.brush( QColorGroup::ButtonText ) );
    } else {
      p->setPen( cg.buttonText() );
      p->setBrush( cg.brush( QColorGroup::Mid ) );
    }
    p->drawPolygon( pa );
  } else {
    // 기초형식이 다른 원시형식을 조종하게 한다.
    QWindowsStyle::drawPrimitive( pe, p, r, cg, flags, data );
  }
}
```

③ 사용자정의형식의 리용

Qt응용프로그람에서 사용자정의형식을 사용하는 몇가지 방법이 있다. 가장 간단한 방법은 응용프로그람의 main()함수에서 다음의 코드행을 포함하는것이다.

```
#include "customstyle.h"
int main( int argc, char ** argv )
{
  QApplication::setStyle( new CustomStyle() );
  // QApplication객체를 창조하는 보통의 루틴.
}
```

또한 자기의 프로젝트에 customstyle.h와 customstyle.cpp 파일들을 포함해야 한다.

④ 플라그가능한 형식의 창조와 리용

자기의 형식을 자기가 작성한 응용프로그람들이나 다른 응용프로그람들에서 재콤파일하지 않고 사용할수 있게 만들려고 할수 있다. Qt Plugin체계는 형식을 플라그인으로 창조할수 있게 한다. 플라그인으로 창조된 형식들은 Qt자체에 의하여 실행시에 공유객체로서 적재된다.

자기의 플라그인을 콤파일하고 그것을 $QTDIR/plugins/styles에 넣는다. 그러면 Qt가 자동적재할수 있는 플라그가능형식을 가지게 된다. 현존 응용프로그람들에서 새로운 형식을 사용하려면 다음의 인수로 응용프로그람을 기동해야 한다.

```
./application -style custom
```

응용프로그람은 자기가 실현한 사용자정의형식으로부터 형식을 사용한다.

제17절. Qt형판서고

Qt형판서고(QTL)는 객체용기들을 제공하는 형판들의 모임이다. 적당한 STL실현을 자기의 모든 목표가동환경에 사용할수 없으면 QTL을 그대신에 사용할수 있다. 이것은 객체들의 목록, 객체들의 벡토르(동적배렬), 서로 관련한 형들의 매프(사전이나 련상배렬이라고도 한다) 그리고 련상반복자들과 알고리듬들을 제공한다. 용기는 다른 객체들을 포함하고 관리하는 객체이며 포함된 객체들을 호출하는 반복자들을 제공한다.

QTL클라스들의 명명관례는 다른 Qt클라스들과 일치한다(실례로 count(), isEmpty()). 또한 그것들은 size()와 empty()와 같은 STL알고리듬들과 호환되는 함수들을 제공한다. 이미 STL map에 습관된 프로그람작성자들은 자기가 좋아한다면 STL호환함수들을 사용할수 있다.

STL과 비교하면 QTL은 오직 STL용기 API의 가장 중요한 특성만 포함하며 비교하면 QTL은 가동환경차이를 가지지 않지만 보통 좀 느리고 적은 목적코드를 전개한다.

보관하려고 하는 객체들의 사본을 만들수 없으면 QPtrCollection과 동료들을 사용해야 하며 그 모두는 값이 아니라 지적자에 대하여 조작한다. 실례로 이것을 QObject로부터 파생된 모든 클라스들에 적용한다. QObject는 복사구성자를 가지지 않으므로 값으로 사용할수 없다. QValueList에 QObject의 지적자들을 보관하는것으로 설정할수 있으나 QPtrList의 직접사용은 이러한 종류의 응용프로그람분야에 더 좋은 선택일수 있다. QPtrCollection에 기초한 다른 모든 용기들처럼 QPtrList는 속도에 관하여 최적화된 값에 기초하는 용기보다도 훨씬 더 옳은 검사를 제공한다.

값의미를 실현하는 객체들이 있고 STL을 자기의 목표가동환경에서 사용할수 없으면 Qt형판서고를 그대신 사용할수 있다. 값의미(value semantics)는 적어도 다음것을 요구한다.

- 복사구성자,
- 대입연산자,
- 기정구성자 즉 인수를 가지지 않는 구성자.

수많은 복사조작이 제기되므로 용기의 좋은 성능을 얻는데서 고속복사구성자가 절대적으로 모든것을 결정한다.

자기 자료를 정렬하려고 한다면 자기 자료의 클라스에 operator<()를 실현해야 한다.

값에 기초한 클라스들의 좋은 후보는 QRect, QPoint, QSize, QString 및 int, bool 혹은 double과 같은 C++기본형들이다.

Qt형판서고는 속도에 맞게 설계되므로 반복자들은 아주 빠르다. 이러한 성능을 달성하기 위하여 QPtrCollection기초용기들에서보다 적은 오유검사를 수행한다. 실례로 QTL용기는 련상된 반복자들을 추적하지 않는다. 이것은 일정한 유효성검사가 (실례로 항목을 삭제하고있을 때) 자동적으로 수행될수 없게 하지만 아주 좋은 성능을 발휘하게 한다.

1. 반복자

Qt형판서고는 지적자가 아니라 값객체들을 취급한다. 그러므로 용기들을 순환하는 수단은 반복자밖에 없다. 이것은 반복자의 크기가 표준지적자의 크기와 일치할 때 결함이 없다.

용기를 순환하려면 다음과 같이 순환을 사용한다.

```
typedef QValueList<int> List;
List list;
for( List::Iterator it = list.begin(); it != list.end(); ++it )
  printf( "Number is %i\n", *it );
```

begin()은 첫 요소를 가리키는 반복자를 돌려주고 end()는 마지막 요소다음을 가리키는 반복자를 돌려준다. end()는 무효한 위치를 표식하므로 그것을 간접참고할수 없다. 이것은 시작점이 begin()이든 fromLast()이든 관계없이 반복에서의 중지조건이다. 최대속도를 위하여 뒤붙이연산자(it++, it--)대신에 현저히 더 빠른 앞붙이연산자(++it, --it)를 가지는 증가 혹은 감소반복자들을 사용한다.

같은 개념을 다른 용기클라스들에 적용할수 있다.

```
typedef QMap<QString,QString> Map;
Map map;
for( Map::iterator it = map.begin(); it != map.end(); ++it )
  printf( "Key=%s Data=%s\n", it.key().ascii(), it.data().ascii() );

typedef QValueVector<int> Vector;
Vector vec;
for( Vector::iterator it = vec.begin(); it != vec.end(); ++it )
  printf( "Data=%d\n", *it );
```

두 종류의 반복자 즉 우의 실례들에서 보여준 volatile반복자와 현재 객체에로의 const참고를 돌려주는 ConstIterator가 있다. const반복자들은 const함수안의 성원변수처럼 용기자체가 const일 때 요구된다. 표준Iterator의 ConstIterator에로의 대입은 그것이 violate const이므로 허용되지 않는다.

2. 알고리듬

Qt형판서고는 그 용기들에 대하여 조작하는 많은 알고리듬을 정의한다. 이러한 알고리듬들은 형판함수들로 실현되며 반복자들을 제공하는 용기(자체의 용기들을 비롯하여)에 적용될수 있는 쓸모있는 범용코드를 제공한다.

① qHeapSort()

qHeapSort()는 잘 알려진 정렬알고리듬을 제공한다. 다음과 같이 그것을 사용할수 있다.

```
typedef QValueList<int> List;
List list;
list << 42 << 100 << 1234 << 12 << 8;
qHeapSort( list );

List list2;
list2 << 42 << 100 << 1234 << 12 << 8;
List::Iterator b = list2.find( 100 );
List::Iterator e = list2.find( 8 );
qHeapSort( b, e );

double arr[] = { 3.2, 5.6, 8.9 };
qHeapSort( arr, arr + 3 );
```

첫 실례는 전체목록을 정렬한다. 둘째 실례는 두 반복자들사이에 떨어지는 요소들 즉 100, 1234, 12만 정렬한다. 셋째 실례는 반복자들이 지적자처럼 작용한다는것과 지적자처럼 취급할수 있다는것을 보여준다.

자체의 자료형을 사용한다면 자기 자료의 클라스에 대하여 operator<()를 실현해야 한다.

정렬형판들은 const반복자들과 작업하지 않는다.

② qSwap()

qSwap()는 두개 변수의 값들을 교환한다.

```
QString second( "Einstein" );
QString name( "Albert" );
qSwap( second, name );
```

③ qCount()

qCount()형판함수는 용기안의 값의 개수를 계수한다. 실례로

```
QValueList<int> list;
list.push_back( 1 );
list.push_back( 1 );
list.push_back( 1 );
list.push_back( 2 );
int c = 0;
qCount( list.begin(), list.end(), 1, c ); // c == 3
```

④ qFind()

qFind()형판함수는 용기안에서 값의 첫 출현을 찾는다. 실례로

```
QValueList<int> list;
list.push_back( 1 );
list.push_back( 1 );
list.push_back( 1 );
list.push_back( 2 );
QValueListIterator<int> it = qFind( list.begin(), list.end(), 2 );
```

⑤ qFill()

qFill()형판함수는 범위를 값들의 사본으로 채운다. 실례로

```
QValueVector<int> vec(3);
qFill( vec.begin(), vec.end(), 99 ); // vec contains 99, 99, 99
```

⑥ qEqual()

qEqual()형판함수는 두개의 범위에 대하여 그 요소들의 등가성을 비교한다. 첫 범위안의 요소들이 둘째 범위안의 대응하는 요소들과 같으면 (따라서 두 범위가 유효이여야 한다.) 매 범위안의 요소수를 고려하지 않는다. 실례로

```
QValueVector<int> v1(3);
v1[0] = 1;
v1[2] = 2;
v1[3] = 3;

QValueVector<int> v2(5);
v2[0] = 1;
v2[2] = 2;
v2[3] = 3;
v2[4] = 4;
v2[5] = 5;

bool b = qEqual( v1.begin(), v2.end(), v2.begin() );
// b == TRUE
```

⑦ qCopy()

qCopy()형판함수는 한 범위의 원소들을 OutputIterator(이 경우에

QTextOStreamIterator)에 복사한다.

```
QValueList<int> list;
list.push_back( 100 );
list.push_back( 200 );
list.push_back( 300 );
QTextOStream str( stdout );
qCopy( list.begin(), list.end(), QTextOStreamIterator(str) );
```

⑧ qCopyBackward()

qCopyBackward()형판함수는 용기 혹은 용기의 일부를 OutputIterator에 반대순서로 복사한다. 실례로

```
QValueVector<int> vec(3);
vec.push_back( 100 );
vec.push_back( 200 );
vec.push_back( 300 );
QValueVector<int> another;
qCopyBackward( vec.begin(), vec.end(), another.begin() );
// 'another' now contains 100, 200, 300
// however the elements are copied one at a time
// in reverse order (300, 200, then 100)
```

⑨ QTL 반복자들

임의의 Qt형판서고반복자를 OutputIterator로 사용할수 있다. 다음의 실례는 이것을 설명한다.

```
QStringList list1, list2;
list1 << "Weis" << "Ettrich" << "Arnt" << "Sue";
list2 << "Torben" << "Matthias";
qCopy( list2.begin(), list2.end(), list1.begin() );

QValueVector<QString> vec( list1.size(), "Dave" );
qCopy( list2.begin(), list2.end(), vec.begin() );
```

이 코드부분의 끝에서 목록 list1은 삽입되는 순서로 "Torben", "Matthias", "Arnt", "Sue"을 포함한다. 벡토르 vec에는 삽입되는 순서로 "Torben", "Matthias", "Dave", "Dave"를 포함된다.

새로운 알고리듬을 쓴다면 될수록 많은 용기에서 사용할수 있도록 써야 한다. 우의 실례에서 qCopy()에 의해 표준C++배렬을 간단히 출력할수 있다.

```
int arr[] = { 100, 200, 300 };
QTextOStream str( stdout );
qCopy( arr, arr + 3, QTextOStreamIterator( str ) );
```

3. 스트림

언급한 모든 용기들은 적당한 스트림연산자들에 의해 계렬화될수 있다. 여기에 실례가 있다.

```
QDataStream str(...);
QValueList<QRect> list;
// ... fill the list here
str << list;
```

용기는 다음과 같은 방법으로 다시 읽어들일수 있다.

```
QValueList<QRect> list;
str >> list;
```

같은 방법을 QStringList, QValueStack, QMap에도 적용할수 있다.

제18절. Qt에서 스레드기능

Qt는 폼에서 Qt기초가동환경에 의존하지 않는 스레드작성클라스들의 스레드유지, 사건들을 발송하는 스레드에 안전한 방법, 여러가지 스레드들로부터 Qt메쏘드들을 호출하게 하는 대역적인 Qt서고잠건을 제공한다.

1. 스레드유지의 허용

Qt를 Windows에 설치할 때 스레드기능는 일부 콤파일러에서는 선택적이다.

Mac OS X와 Unix에서 스레드유지는 configure스크립트를 실행시킬 때 -thread 선택을 추가함으로써 허용할수 있다. 다중스레드프로그람들이 특수한 libc와 같이 특별한 방법으로 련결되여야 하는 Unix가동환경에서 설치과정에 개별적인 서고libqt-mt를 창조하므로 스레드프로그람들은 표준Qt서고가 아니라 이 서고에 대하여(-lqt-mt에 의해) 련결되여야 한다.

두 가동환경에서 정의된 QT_THREAD_SUPPORT마크로를 리용하여 콤파일하여야 한다.(실례로 -DQT_THREAD_SUPPORT로 콤파일한다.) Windows에서 이것은 보통 qconfig.h안의 항목에 의하여 수행된다.

2. 스레드클라스들

이 클라스들은 스레드유지를 허용할 때 Qt서고안에 만들어진다.

- QThread - 새 스레드를 기동하는 수단들을 제공하며 새 스레드는 QThread::run()재정의에서 실행을 시작한다. 이것은 Java스레드클라스와 비슷하다.
- QThreadStorage - 스레드마다 자료기억기를 제공한다. 이 클라스는 QThread에 의해 기동된 스레드에만 사용될수 있으며 가동환경에 고유한 API에 의해 기동되는 스레드에 사용할수 없다.
- QMutex - 호상배제형 잠건(뮤텍스)을 제공한다.
- QMutexLocker - 자동적으로 QMutex를 잠건하거나 여는 편의클라스. QMutexLocker는 복잡한 코드나 례외를 사용하는 코드에서 사용할수 있다.
- QWaitCondition - 다른 스레드에 의하여 작업할 때까지 스레드를 일시 중지하는 방법을 제공한다.
- QSemaphore - 간단한 옹근수쎄마퍼를 제공한다.

3. 중요한 정의들

다중스레드프로그람에서 Qt를 사용할 때 재입구가능과 스레드안전이라는 용어들을 리해하는것이 중요하다.

- 재입구가능(*reentrant*) - 함수호출시마다 유일한 자료를 참고하는 다중스레드들에 의하여 동시에 호출될수 있는 함수를 서술한다. 재입구가능함수를 동시에 같은 이름으로 호출하는것은 불안전하며 이와 같은 호출은 련속적이여야 한다.
- 스레드안전(*thread-safe*) - 호출시마다 공유자료를 호출할 때 다중스레드에 의하여 동시에 호출될수 있는 함수를 서술한다. 스레드안전함수를 동시에 같은 자료를 가지고 호출하는것은 안전하므로 공유자료에로의 모든 접근은 련속적이다.

Qt는 암시적으로 공유된 클라스들과 명시적으로 공유된 클라스들을 둘다 제공한다.

대부분의 C++성원함수들은 선천적으로 재입구가능하므로 오직 클라스성원자료들을 참고할뿐이다. 하나의 스레드는 다른 어떤 스레드도 한 실례에 대하여 성원함수를 호출

하지 않는한 같은 실례에 대하여 그러한 성원함수를 호출할수 있다. 실례로 클라스 Number를 주었다.

```
class Number
{
public:
  inline Number( int n ) : num( n ) { }

  inline int number() const { return num; }
  inline void setNumber( int n ) { num = n; }

private:
  int num;
};
```

메쏘드 Number::number()와 Number::setNumber()는 재입구가능이므로 유일한 자료를 참고한다. Number의 매 실례에 대하여 한번에 오직 하나의 스레드만이 성원함수들을 호출할수 있다. 하지만 다중스레드는 Number의 개별적인 실례에 대하여 성원함수를 호출할 수 있다.

보통 스레드안전함수들은 뮤텍스(실례로 QMutex)를 리용하여 공유자료에 대한 호출을 계렬화한다. 뮤텍스의 잠건과 열기의 추가비용으로 인하여 스레드안전함수들은 보통 재입구가능함수들보다 느리다. 실례로 아래에 클라스 Counter를 주었다.

```
class Counter
{
public:
  inline Counter()  { ++instances; }
  inline ~Counter() { --instances; }

private:
  static int instances;
};
```

정적인 instances옹근수의 수정은 계렬화되지 않으므로 이 클라스는 스레드에 안전하지 않다. 그러므로 그것을 스레드안전하게 만들고 뮤텍스를 사용해야 한다.

```
class Counter
{
public:
  inline Counter()
  {
    mutex.lock();
    ++instances;
    mutex.unlock();
  }
  ...
private:
  static QMutex mutex;
  static int instances;
};
```

4. 스레드에 안전한 사건발송

Qt에서 하나의 스레드는 항상 도형사용자대면부이거나 사건스레드이다. 이것은 QApplication객체를 창조하고 QApplication::exec()를 호출하는 스레드이다. 이것은 또한 프로그람이 기동할 때 main()을 호출하는 초기스레드이다. 이 스레드는 창문체계로부터 사건의 생성과 수신을 비롯하여 도형사용자대면부조작을 수행하게 하는 유일한 스레드이다. Qt는 두번째 스레드에서 QApplication창조와 QApplication::exec()에 의한 사건순환고리실행을 유지하지 못한다. QApplication을 창조하고 자기의 프로그람에서 main()함수로부터 QApplication::exec()를 호출하여야 한다.

창문부품의 자료를 현시하려고 하는 스레드들은 직접 그 창문부품을 수정할수 없으므로 QApplication::postEvent()를 사용하여 그 창문부품을 사건에 발송하여야 한다. 그 사건은 도형사용자대면부스레드에 의하여 후에 발송된다.

보통 프로그람작성자는 창문부품에 보내온 사건에서 어떤 정보를 포함하려고 한다. (사용자정의사건들에 대한 자세한 정보는 QCustomEvent문서를 참고하시오.)

5. 스레드와 QObject파생클라스들

QObject클라스그자체는 재입구가능이지만 도형방식사용자대면부스레드가 아닌 스레드에서 QObject를 창조하고 사용할 때 일정한 규칙들을 적용한다.

① Qt서고에 포함된 어떤 QObject기초클라스도 재입구가능이다. 이것은 모든 창문부품(실례로 QWidget와 그 파생클라스), 조작체계핵심클라스(실례로 QProcess, QAccel, QTimer), 모든 망구축클라스(실례로 QSocket, QDns)들을 포함한다.

② QObject와 그의 모든 파생클라스들은 스레드에 안전하지 않다. 이것은 전체 사건송달체계를 포함한다. GUI스레드는 자기의 QObject파생클라스에 사건들을 송달하는 동안 다른 스레드로부터 그 객체를 호출하고있다는것을 기억해두는것이 중요하다. 비GUI스레드에서 QObject를 사용하고있고 이 객체에 보내온 사건들을 조종하고있다면 뮤텍스를 가지고 자기 자료에로의 모든 접근을 보호해야 한다.

③ 종속된 사건들이 송달을 기다리는 동안 QObject의 삭제는 중단을 일으킬수 있다. 비GUI스레드로부터 직접 QObject를 삭제하지 말아야 한다. 그대신 QObject::deleteLater()메쏘드를 사용하여야 한다. 그러면 종속된 모든 사건들이 객체에 송달된 다음에 사건순환고리가 객체를 삭제하게 한다.

6. Qt서고 뮤텍스

QApplication은 창문체계함수에 대한 접근을 보호하는데 쓰이는 뮤텍스를 포함한다. 이 뮤텍스는 사건순환고리를 실행하는동안(실례로 사건을 송달하는 기간) 잠건되고 사건순환고리가 일시 중지될 때 잠건이 해제된다.

알아두기: Qt사건순환고리는 재귀적이고 서고뮤텍스는 사건순환고리에 재입구할 때(실례로 QDialog::exec()에 의해 이행금지대화칸을 실행할 때) 잠건이 해제된다.

다른 스레드가 Qt서고뮤텍스를 잠건할 때 사건순환고리는 사건처리를 중지하며 잠건하는 스레드는 단순한 GUI조작을 수행할수 있다. QPainter의 창조와 선분그리기와 같은 조작은 단순한 GUI조작의 실례이다.

```
...
qApp->lock();

QPainter p;
p.begin( mywidget );
p.setPen( QColor( "red" ) );
p.drawLine( 0,0,100,100 );
```

```
p.end();

qApp->unlock();
...
```

사건들을 생성하는 조작들은 어떤 비GUI스레드에 의하여 호출되지 말아야 한다. 그러한 조작들의 실례들은 다음과 같다.

- QWidget, QTimer, QSocketNotifier, QSocket 혹은 다른 망클라스들의 창조.
- QWidget의 이동, 크기조절, 표시 혹은 숨기기.
- QTimer의 시작이나 중지.
- QSocketNotifier의 허용과 금지.
- QSocket 혹은 다른 망클라스의 사용.

일부 가동환경에서는 이 조작들에 의해 생성된 사건들을 잃는다.

7. 스레드와 신호, 처리부들

QObject기초클라스를 위한 규칙들을 따르는 한 신호-처리부기구를 개별적인 스레드에서 사용할수 있다. 신호-처리부기구는 동기적이고 신호가 발생될 때 모든 처리부는 즉시 호출된다. 처리부는 신호를 발생한 스레드문맥에서 실행된다.

경고: 창문체계사건들을 생성하거나 창문체계함수들을 사용하는 처리부들은 비GUI스레드로부터 발생되는 신호와 련결되지 말아야 한다.

8. 스레드와 공유자료

Qt는 많은 암시적공유클라스와 명시적공유클라스들을 제공한다. 다중스레드프로그람에서 공유클라스의 여러개의 실례들이 공유자료를 참고할수 있는데 하나이상의 스레드들이 그 자료를 수정하려고 시도한다면 위태로워진다. Qt는 QDeepCopy클라스를 제공하는데 이 클라스는 공유클라스들이 유일자료를 참고하도록 담보한다(15절 참고).

9. 스레드와 SQL모듈

련결은 그것을 창조한 스레드안에서만 사용될수 있다. 스레드들사이에서 련결의 옮기기와 다른 스레드로부터 질문의 창조는 유지되지 않는다.

또한 QSqlDriver들에 의하여 사용된 제3자가 제공하는 서고들은 다중스레드프로그람에서 SQL모듈사용에 관한 다른 제한을 강요할수 있다.

10. 경고

스레드프로그람을 작성할 때 주의하여야 할 점

- 우에서 언급한것처럼 QObject기초클라스들은 스레드안전도 재입구가능도 아니다. 이것은 모든 창문부품들(실례로 QWidget와 그 파생클라스들)과 조작체계핵심클라스(실례로 QProcess, QAccel), 모든 망클라스(실례로 QSocket, QDns)들을 포괄한다.
- 종속된 사건들이 송달되기를 기다리는동안 QObject의 삭제는 중단을 일으킨다. GUI스레드가 아닌 스레드에서 QObject들을 창조하고 이 객체들에 사건을 발송하고있다면 QObject를 직접 삭제하지 말아야 한다. 그대신 QObject::deleteLater()메쏘드를 사용하여 모든 종속된 사건들을 객체에 발송한 다음에 사건순환고리가 그 객체를 삭제하게 한다.
- Qt서고뮤텍스를 보유하고있는동안 폐색조작을 수행하지 말아야 한다. 이것은 사건순환고리를 동결시킨다.
- 재귀적인 QMutex를 잠건한 회수만큼 열고있는가 확인하여야 한다. 이것은 적어도 안되고 많아도 안된다.
- 자기 응용프로그람에서 보통의 Qt서고와 스레드화된 Qt서고를 섞지 말아야 한다.

이것은 자기 응용프로그람이 스레드화된 Qt서고를 사용한다면 보통의 Qt서고와 련결하지 말고 보통의 Qt서고를 동적으로 적재하거나 Qt서고에 의존하는 다른 서고나 플래그인을 동적으로 적재하여야 한다. 일부 체계에서는 Qt서고에서 사용되는 정적자료를 못쓰게 할수 있다.

• Qt는 둘째 스레드에서 QApplication의 창조와 사건순환고리의 실행(QApplication::exec()에 의해)을 지원하지 않는다. 자기 프로그람에서 QApplication객체를 창조하고 main()함수로부터 QApplication::exec()를 호출해야 한다.

제19절. 시계

모든 Qt객체들의 기초클라스인 QObject는 Qt의 기초시계기능을 제공한다. QObject::startTimer()에서 인수로서 ms의 시격을 가지는 시계를 기동한다. 함수는 유일한 옹근수인 시계id를 돌려준다. 현재 시계는 시계id를 가지고 명백히 QObject::killTimer()를 호출할 때까지 매 시격마다 《발화》된다.

이 기구가 작업하려면 응용프로그람은 사건순환고리에서 실행되여야 한다. QApplication::exec()를 가지고 사건순환고리를 시작한다. 시계가 발화될 때 응용프로그람은 QTimerEvent를 송신하고 조종의 흐름은 시계사건이 처리될 때까지 사건순환고리를 떠난다. 이것은 자기 응용프로그람이 다른 일로 분주할 때 시계를 발화할수 없다는 것을 암시한다. 다시 말하면 시계의 정확도는 자기 응용프로그람의 립도(granularity)에 의존한다.

실천적으로 시격값에 대한 상한은 없다. (1년이상도 가능하다.) 정확도는 기초하고 있는 조작체계에 의존한다. Windows 95/98는 55ms (s당 18.2회)의 정확도를 가진다. 다른 체계들(UNIX X11와 Windows NT)은 1ms의 시격을 조종할수 있다.

시계기능의 기본API는 QTimer이다. 이 클라스는 시계가 발화할 때 신호를 발생하는 정규시계를 제공하며 대부분의 GUI프로그람들의 소유자구조에 아주 적합하도록 QObject를 계승한다. 그것을 사용하는 표준방법은 다음과 같다.

```
QTimer * counter = new QTimer( this );
connect( counter, SIGNAL(timeout()), this,
SLOT(updateCaption()) );
counter->start( 1000 );
```

계수기시계는 이 창문부품의 자식으로 만들어지므로 창문부품이 삭제될 때 시계도 삭제된다. 다음에 그 시간요구신호는 작업을 수행할 처리부에 련결되고 끝으로 그것이 기동된다.

또한 QTimer는 단순한 한번발사시계API를 제공한다. QButton은 이것을 사용하여 눌리운 단추를 표시하며 건반으로 단추를 누른 경우에 0.1s후에 단추를 놓는다. 실례로

```
QTimer::singleShot( 100, this, SLOT(animateTimeout()) );
```

이 코드행의 실행 0.1s후에 같은 단추의 animateTimeout()처리부가 호출된다.

여기에 신호와 처리부를 통한 객체통신을 QTimer객체와 결합하는 현저히 긴 실례의 륜곽을 보여준다. 이것은 사용자대면부를 페색하지 않고 단일스레드응용프로그람에서 긴장한 계산을 수행하는데 시계를 사용하는 방법을 보여준다.

```
// The Mandelbrot class uses a QTimer to calculate the mandelbrot
// set one scanline at a time without blocking the CPU. It
// inherits QObject to use signals and slots. Calling start()
```

// starts the calculation. The done() signal is emitted when it
// has finished. Note that this example is not complete, just an outline.

```
class Mandelbrot : public QObject
{
  Q_OBJECT // required for signals/slots
public:
  Mandelbrot( QObject *parent=0, const char *name );
  ...
public slots:
  void start();
signals:
  void done();
private slots:
  void calculate();
  private:
  QTimer timer;
  ...
};

// Constructs and initializes a Mandelbrot object.
Mandelbrot::Mandelbrot( QObject *parent=0, const char *name )
: QObject( parent, name )
{
  connect( &timer, SIGNAL(timeout()), SLOT(calculate()) );
  ...
}

// Starts the calculation task. The internal calculate() slot
// will be activated every 10 milliseconds.

void Mandelbrot::start()
{
  if ( !timer.isActive() ) // not already running
    timer.start( 10 );   // timeout every 10 ms
}

// Calculates one scanline at a time.
// Emits the done() signal when finished.

void Mandelbrot::calculate()
{
  ...        // perform the calculation for a scanline
  if ( finished ) {  // no more scanlines
    timer.stop();
```

```
        emit done();
      }
    }
```

제20절. Qt창문부품들의 화상

대부분의 창문부품들을 Motif 혹은 Windows형식으로 표시한다. 모든 창문부품들은 두 형식(그리고 다른 형식)으로 유지되지만 명백성때문에 바로 선택을 표시한다.

Qt는 아래에 보여준 선택이상으로 창문부품들을 제공한다. 그리고 여분의 Qt를 제3자로부터도 사용할수 있으며 Qt에 대한 추가로서 사용할수도 있다.

그림 4-3에 QSplitter에 의해 분리된 3개의 보기가 있다. 왼쪽우에 QListBox가 있고 오른쪽우에 하나의 QHeader와 두개의 QScrollBar를 가진 QListView가 있다. 그리고 아래에 QIconView가 있다.

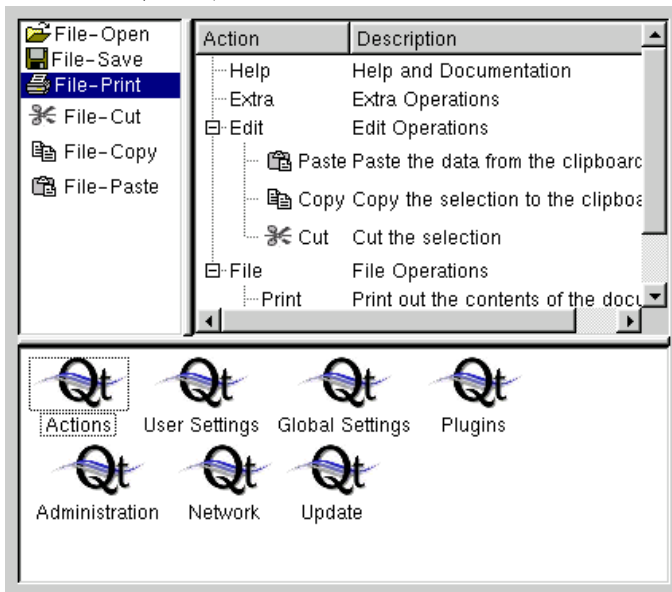



그림 4-3. QSplitter에 의해 분리된 3개의 보기

그림 4-4에 하나의 QMenuBar 그리고 QToolButton들과 QComboBox와 같은 각종 창문부품들을 포함하는 QToolBar들을 가지는 QMainWindow를 제시한다. 중심창문부품은 MDI창문관리에 쓰이고 QTextEdit를 특징짓는 MDI창문을 포함하는 QWorkspace이다. 바닥에 하나의 QStatusBar가 있고 오른쪽밑에 하나의 QSizeGrip가 있다.

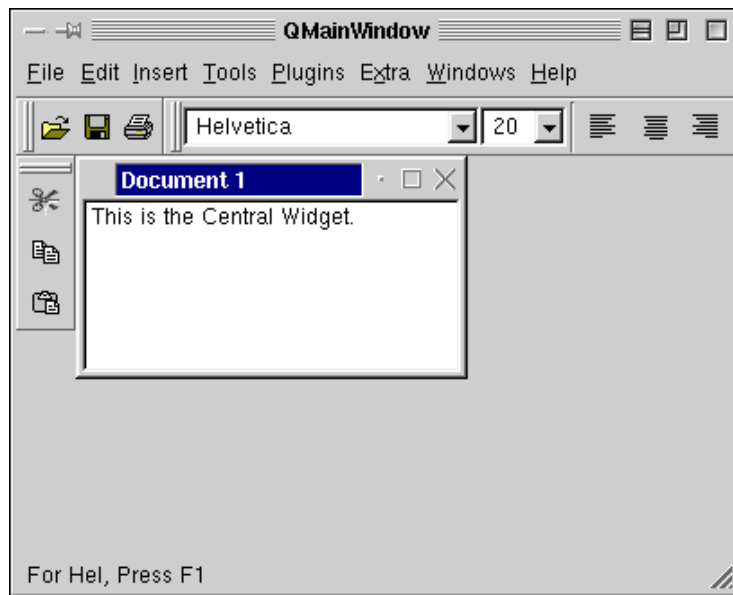



그림 4-4. QMainWindow

아래의 화상은 QFileDialog를 보여준다. Macintosh와 Windows가동환경에서 QFileDialog를 사용하거나 본래의 파일대화칸을 사용할수 있다. 이것은 QFileDialog 클라스문서에서 설명한다.

그림 4-5. QFileDialog

그림 4-6에 QPrintDialog가 있다. Macintosh와 Windows에서는 본래의 인쇄대화칸이 쓰이지만 다른 가동환경에서는 QPrintDialog를 제공한다. 가동환경에 의존하지 않게 하려면 이식성을 위하여 QPrintDialog대신에 QPrinter::setup()를 사용한다.

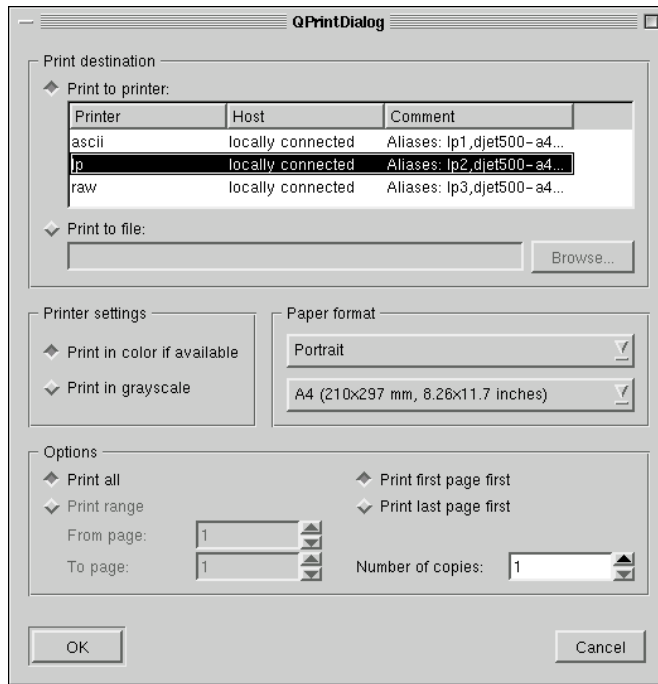



그림 4-6. QPrintDialog

그림 4-7에 QFontDialog가 있다.

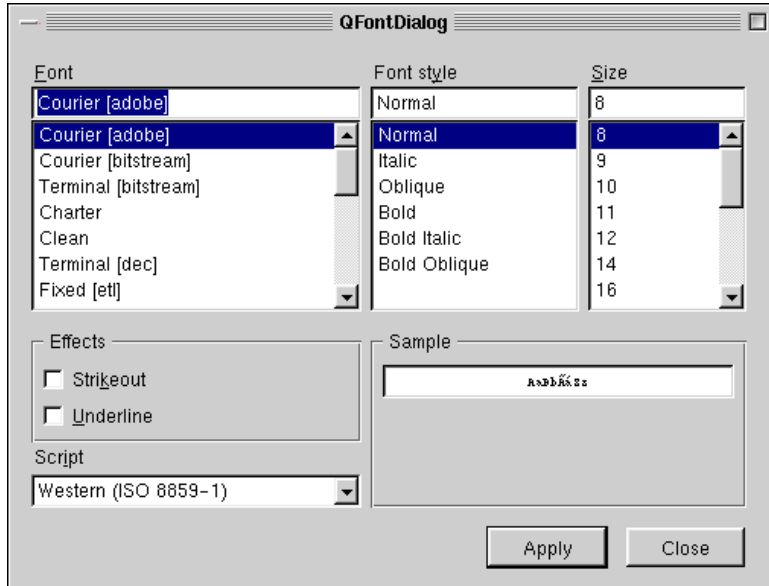



그림 4-7. QFontDialog

그림 4-8에 QColorDialog를 보여준다.

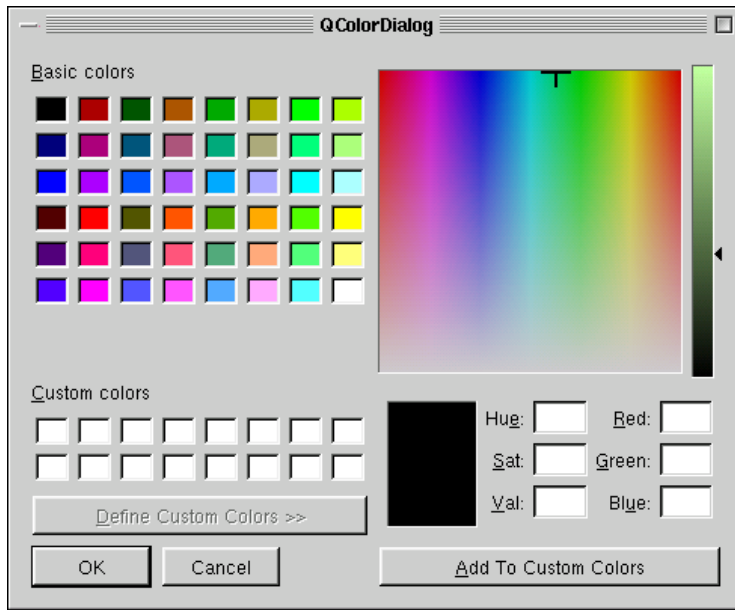



그림 4-8. QColorDialog

그림 4-9에 보여주는것처럼 통보문들은 QMessageBox에 의해 표시된다.

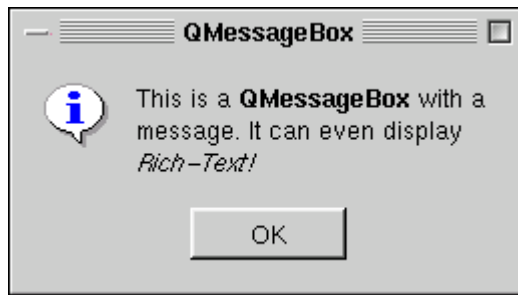



그림 4-9. QMessageBox

그림 4-10은 QProgressDialog를 표시한다. QProgressBar도 역시 개별적창문부품으로서 사용될수 있다.

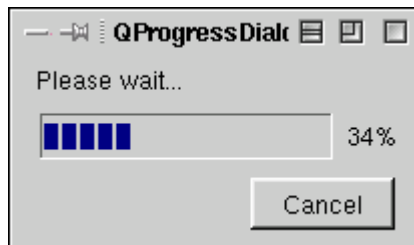



그림 4-10. QProgressDialog

그림 4-11에는 하나의 QLineEdit, 하나의 읽기전용QComboBox 및 편집가능한 하나의 QComboBox를 포함하는 QGroupBox가 있다.

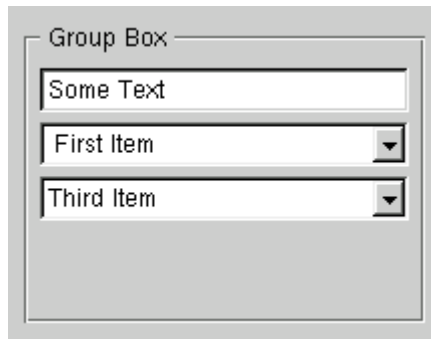



그림 4-11. QGroupBox

그림 4-12는 QPopupMenu를 보여준다.

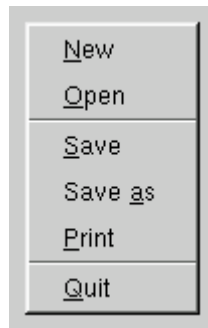



그림 4-12. QPopupMenu

그림 4-13에 4개의 QRadioButton와 두개의 QCheckBox들을 포함하는 QButtonGroup가 있다.

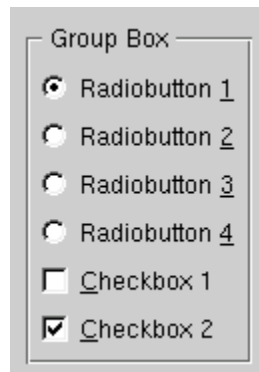



그림 4-13. QButtonGroup

그림 4-14는 QTabDialog를 보여준다. 타브 (QTabBar) 혹은 하나의 타브띠와 페지들을 결합하는 더 편리한 클라스QTabWidget가 개별적으로 사용될수 있다. 페지에서 하나의 QLabel, 범위조종요소들인 QSlider와 QSpinBox, 그리고 아래에 하나의 QLCDNumber가 보인다. 바닥에 여러개의 QPushButton이 있다.

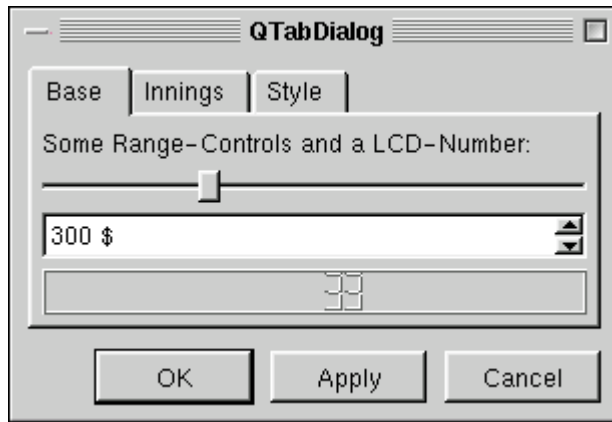



그림 4-14. QTabDialog

그림 4-15에 HTML페지를 현시하는 QTextBrowser가 있다.

그림 4-15. QTextBrowser

제21절. Qt응용프로그람의 전개

이 절에서는 Qt응용프로그람들을 배포하는데 필요한 가동환경에 고유한 파일들을 보여준다. 역시 요구될수 있는 콤파일러에 고유한 파일들을 포함하지 않는다(8장 1절 참고).

1. 정적인 Qt응용프로그람

정적인 Qt응용프로그람을 배포하려면 모든 가동환경에 대하여 자기 응용프로그람의 실행파일이 필요하다.

2. 동적인 Qt응용프로그람

동적인 Qt응용프로그람을 배포하려면 모든 가동환경에 대하여 응용프로그람의 실행

파일과 Qt서고파일들이 필요하다.

Qt서고는 응용프로그람의 실행파일과 같은 등록부에, 또는 체계서고경로에 포함되는 등록부에 있어야 한다.

서고는 표 4-7과 같은 가동환경에 고유한 파일들에 의하여 제공된다.

표 4-7. 가동환경에 고유한 Qt서고파일

가동환경	파일
Windows	qt[version].dll
Unix/Linux	libqt[version].so
Mac	libqt[version].dylib

*version*은 3개의 판번호를 포함한다. 스레드구축일 때 판앞에는 앞붙이 *-mt*가 붙는다.

- 플라그인의 배포

응용프로그람이 요구하는 플라그인파일들을 포함해야 한다.

플라그인들은 Qt에 플라그인등록부로 알려진 등록부아래의 보조등록부에 넣어야 한다. 보조등록부는 플라그인범주의 이름을 가져야 한다. (실례로 styles, sqldrivers, designer 등.)

Qt는 다음의 등록부들에서 플라그인범주를 검색한다.

- 응용프로그람에 고유한 플라그인경로
- Qt의 구축등록부
- 응용프로그람등록부

응용프로그람에 고유한 플라그인경로는 QApplication::addLibraryPath()에 의하여 추가할수 있다. Qt의 구축등록부는 Qt서고에 코드로 들어있으며 설치과정의 일부로서 변경될수 있다.

3. 동적대화칸

동적대화칸에서 QWidgetFactory를 사용한다면 모든 가동환경들에 다음과 같은 파일들이 요구된다.

- 동적Qt응용프로그람들에 사용된것과 같은 파일들
- QUI서고

QUI서고는 표 4-8과 같은 가동환경에 고유한 파일들에 의하여 제공된다.

표 4-8. 가동환경에 고유한 QUI서고파일

가동환경	파일
Windows	qui.lib
Unix/Linux	libqui.so
Mac	libqui.dylib

QUI서고는 응용프로그람의 실행파일과 같은 등록부에 있거나 체계서고경로에 포함되는 등록부에 있어야 한다.

제5장. 기하학적배치

제1절. 배치관리자클라스

Qt배치체계는 자식창문부품들의 배치를 지정하는 단순하고도 강력한 방법을 제공한다. 론리적배치를 지정함으로써 다음의 리득을 얻는다.

① 자식창문부품들의 위치지정.
② 제일 웃준위 창문부품들을 위한 감별할수 있는 기정크기들.
③ 제일 웃준위 창문부품들을 위한 감별할수 있는 최소크기들.
④ 크기조종.
⑤ 내용이 변할 때 자동갱신.

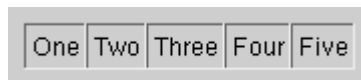
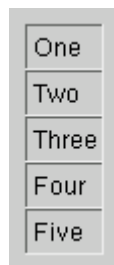
- 보조창문부품들의 서체크기, 본문 혹은 기타 내용들.
- 보조창문부품의 은폐 혹은 표시.
- 보조창문부품들의 삭제.

Qt의 배치관리자클라스들은 손으로 작성하는 C++코드용으로 설계되였으므로 리해와 사용이 간단하다.

손으로 쓴 배치코드의 결함은자기가 폼설계를 실험하고있을 때와 변경시마다 콤파일, 련결, 실행해야 하는 경우에 편리하지 못한것이다. 해결대책은 Qt Designer를 사용하는것으로서 이것은 배치를 간단히 고속으로 하게 하고 C++배치코드를 생성하는 시각적인 GUI설계도구이다 .

1. 배치관리자창문부품

자기 창문부품들에 좋은 배치를 주는 가장 간단한 방법은 배치관리자창문부품 즉 QHBox, QVBox, QGrid를 사용하는것이다. 배치관리자창문부품은 자식창문부품들이 구성되는 순서로 그것들을 자동적으로 배치한다. 더 복잡한 배치를 창조하려면 서로 내부에 배치관리자창문부품들을 겹쌓을수 있다. (QWidget는 기정적으로 배치관리자를 가지지 않으며 QWidget안에서 창문부품들을 배치하려고 한다면 그것을 추가해야 한다.)

- QHBox는 수평행안에 있는 자식창문부품들을 왼쪽에서 오른쪽으로 배치한다.

- QVBox는 수직렬안에 있는 자식창문부품들을 우에서 아래로 배치한다.

- QGrid는 자식창문부품들을 2차원살창형태로 배치한다. 살창이 가지는 렬수를 지정할수 있으며 앞의 행이 다 차면 새로운 행을 시작하면서 왼쪽에서 오른쪽으로 옮겨진다. 살창은 고정되고 자식창문부품들은 창문부품의 크기가 조절될 때 다른 행들로 흐르지 않는다.

우에 보여준 살창은 다음의 코드에 의해 생성될수 있다.

```
QGrid *mainGrid = new QGrid( 2 ); // 2×n살창
new QLabel( "One", mainGrid );
new QLabel( "Two", mainGrid );
new QLabel( "Three", mainGrid );
new QLabel( "Four", mainGrid );
new QLabel( "Five", mainGrid );
```

자식창문부품들에 대하여 QWidget::setMinimumSize() 혹은 QWidget::setFixedSize()를 호출하여 일정한 범위까지 배치를 조절할수 있다.

2. 배치관리자에 창문부품들의 추가

배치관리자에 창문부품들을 추가하는 배치과정은 다음과 같다.

① 모든 창문부품들은 처음에 그것들의 QWidget::sizePolicy()에 따라서 일정한 크기의 공간에 할당된다.

② 창문부품들이 0이상의 값을 가지는 신축결수설정을 가진다면 신축결수에 비례하여 공간에 할당된다

③ 일부 창문부품들은 공간을 요구하지 않고 다른 창문부품들은 신축결수설정이 0이라면 더 큰 공간을 차지한다. 이것들에 대하여 우선 크기확장정책에 따라 창문부품들에 공간이 할당된다.

④ 최소크기(혹은 최소크기가 지정되지 않으면 최소크기암시)이하의 공간이 할당되는 창문부품들에는 최소크기가 할당된다. (창문부품들은 신축결수가 그것들의 결정인자(determining factor)인 경우에 최소크기 혹은 최소크기암시를 가지지 말아야 한다.

⑤ 자기의 최대크기이상 공간에 할당되는 창문부품들은 필요한 최대크기공간에 할당된다. (창문부품들은 신축결수가 자기의 결정인자인 경우에 최대크기를 가지지 말아야 한다.)

- 신축결수

창문부품들은 보통 신축결수(stretch factor)설정없이 창조된다. 그것들이 어떤 배치관리자안에서 배치될 때 창문부품들은 그들의 QWidget::sizePolicy() 혹은 최소크기암시중 큰 값에 따라서 일정한 비률의 공간에 주어진다. 신축결수들은 창문부품들에 호상비례하여 주어지는 공간의 량을 변경하는데 쓰인다.

신축결수설정없이 QHBox를 사용하는 3개의 창문부품을 배치하였다면 다음과 같은 배치를 얻는다.

매개 창문부품에 신축결수들을 적용한다면 그것들은 균형적으로 배치된다. (그러나 그것들의 최소크기암시보다 절대로 작을수 없다.) 실례로

3. QLayout의 파생클라스화

배치관리자를 더 조종할 필요가 있다면 QLayout파생클라스를 사용해야 한다. Qt에 포함된 배치관리자클라스들은 QGridLayout와 QBoxLayout이다. (QHBoxLayout와 QVBoxLayout는 사용하기 쉽고 코드를 읽기 쉽게 하는 QBoxLayout의 보통파생클라스이다.)

배치관리자를 사용할 때 매개 자식을 그 부모창문부품과 배치관리자에 둘다 삽입해야 한다. (보통 구성자에서 addWidget()함수를 호출하여 수행한다.) 이렇게 매개 창문부품에 대하여 정렬(alignment), 신축, 배치와 같은 속성들을 지정하여 배치관리자 파라메터들을 줄수 있다.

다음의 코드는 우의 코드에서 두가지 개량하여 살창을 만든다.

```
QWidget *main = new QWidget;

// 1×1살창을 만든다. 이것은 자동전개된다.
QGridLayout *grid = new QGridLayout( main, 1, 1 );

// (행, 렬) 주소화로 처음 4개 창문부품을 추가한다.
grid->addWidget( new QLabel( "One", main ),   0, 0 );
grid->addWidget( new QLabel( "Two", main ),   0, 1 );
grid->addWidget( new QLabel( "Three", main ), 1, 0 );
grid->addWidget( new QLabel( "Four", main ),  1, 1 );

// 행2, 렬0-렬1, 중심에 마지막창문부품을 추가한다.
grid->addMultiCellWidget( new QLabel( "Five", main ), 2, 2, 0, 1,
Qt::AlignCenter );

// 렬0과 1의 폭들사이의 비가 2:3되게 한다.
grid->setColStretch( 0, 2 );
grid->setColStretch( 1, 3 );
```

구성자에서 파라메터로서 부모배치관리자를 제공하여 배치관리자안에 배치관리자들을 삽입할수 있다.

```
QWidget *main = new QWidget;
QLineEdit *field = new QLineEdit( main );
QPushButton *ok = new QPushButton( "OK", main );
QPushButton *cancel = new QPushButton( "Cancel", main );
QLabel *label = new QLabel( "Write once, compile everywhere.",
main );

// 창문부품우의 배치관리자
QVBoxLayout *vbox = new QVBoxLayout( main );
vbox->addWidget( label );
vbox->addWidget( field );

// 배치관리자안의 배치관리자
QHBoxLayout *buttons = new QHBoxLayout( vbox );
buttons->addWidget( ok );
```

buttons->addWidget(cancel);

기정배치가 만족되지 않으면 부모없이 배치관리자를 창조한 다음 addLayout()에 의해 그것을 삽입한다. 그때 안쪽 배치관리자는 삽입되는 배치관리자의 자식으로 된다.

4. 사용자정의배치관리자

기본배치관리자클라스들이 충분하지 않으면 자체로 정의할수 있다. 자기의 배치관리자클라스를 순환하는 QGLayoutIterator의 파생클라스는 물론 크기를 조절하고 크기를 계산하는 QLayout의 파생클라스를 만들어야 한다.

5. 배치관리자안의 사용자정의창문부품

또한 자체의 창문부품클라스를 만들 때 그의 배치관리자속성들과 교제해야 한다. 창문부품이 QLayout를 가진다면 이것은 이미 고려된다. 창문부품에 자식창문부품이 없거나 수동배치관리자를 사용한다면 다음과 같은 QWidget성원함수들을 재정의해야 한다.

- QWidget::sizeHint()는요구되는 창문부품의 크기를 돌려준다.
- QWidget::minimumSizeHint()는창문부품이 가질수 있는 최소크기를 돌려준다.
- QWidget::sizePolicy()는창문부품의 공간요구를 서술하는 값인 QSizePolicy를 돌려준다.

크기암시, 최소크기암시 혹은 크기정책이 변할 때 QWidget::updateGeometry()를 호출한다. 이것은 배치를 다시 하게 한다. updateGeometry()에 대한 다중호출은 오직 한번 재계산하게 한다.

요구되는 창문부품의 높이가 그 실제너비(실례로 자동단어중지를 가지는 표식)에 의존한다면 sizePolicy()의 hasHeightForWidth()기발을 설정하고 QWidget::heightForWidth()를 재정의한다.

heightForWidth()를 재정의하여도 아직 좋은 sizeHint()를 제공할 필요가 있다. sizeHint()는 창문부품이 요구하는 폭을 제공하며 그것은 heightForWidth()를 지원하지 않는 QLayout파생클라스(QGridLayout와 QBoxLayout는 그것을 지원한다.)에서 사용된다

이 함수들의 실현에 대한 자세한 차림표는 자기의 새 창문부품에 대하여 류사한 배치요구를 가지는 현존 Qt클라스들에서 그 실현방법을 참고하시오.

6. 수동배치관리자

일종의 특수배치관리자를 만들고있다면 우에 서술한 사용자정의창문부품을 만들수도 있다. QWidget::resizeEvent()를 재정의하여 필요한 크기분배량을 계산하고 매개자식에 대하여 setGeometry()를 호출한다.

창문부품은 배치를 재계산해야 할 때 LayoutHint형의 사건을 얻는다. QWidget::event()를 재정의하여 LayoutHint사건들에 대하여 통지한다.

7. 배치문제

표식자창문부품에서 리치본문의 사용은 그 부모창문부품의 배치에 문제를 일으킬수 있다. 표식자에서 단어의 일부가 가리울 때 리치본문을 Qt의 배치관리기가 조종하는것으로 인하여 문제가 발생한다. 일정한 경우에 부모배치관리자는 QLayout::FreeResize방식으로 넣어지는데 이것은 작은 크기의 창문들에 맞게 내용의 배치를 받아들이지 않거나 사용자가 창문을 사용할수 없을 정도로 너무 작게 만드는것을 방지한다는것을 의미한다. 이것은 문제를 일으킨 창문부품들의 파생클라스를 만들고 적당한 sizeHint()와 minimumSizeHint()함수들을재정의하여 극복할수 있다.

제2절. 자체의 배치관리기작성

여기서는 하나의 실례를 구체적으로 제시한다. 클라스 CardLayout는 같은 이름의 Java배치관리기에 받아들일수 있다. 그것은 항목들(창문부품이나 겹쌓인 배치관리자들)을 서로의 우에 배치한다. 이때 매개 항목은 QLayout::spacing()만큼씩 변위된다.

자체의 배치관리자클라스를 쓰려면 다음과 같이 정의해야 한다.

배치관리자에 의해 조절된 항목들을 보관하는 자료구조. 매개 항목은 QLayoutItem이다. 실례에서 QPtrList를 사용한다.

- addItem(), 배치관리자에 항목들을 추가한다.
- setGeometry(), 배치를 진행한다.
- sizeHint(), 배치관리자의 등록된 크기.
- iterator(), 배치관리자에 대하여 반복한다.

대부분의 경우에 minimumSize()도 실현한다.

(1) card.h

```
#ifndef CARD_H
#define CARD_H

#include <qlayout.h>
#include <qptrlist.h>

class CardLayout : public QLayout
{
public:
  CardLayout( QWidget *parent, int dist ) : QLayout( parent, 0, dist ) {}
  CardLayout( QLayout* parent, int dist) : QLayout( parent, dist ) { }
  CardLayout( int dist ) : QLayout( dist ) {}
  ~CardLayout();

  void addItem(QLayoutItem *item);
  QSize sizeHint() const;
  QSize minimumSize() const;
  QLayoutIterator iterator();
  void setGeometry(const QRect &rect);

private:
  QPtrList<QLayoutItem> list;
};
#endif
```

(2) card.cpp

```
#include "card.h"
```

우선 배치관리자에 대한 반복자를 정의한다. 배치관리자반복자는 창문부품삭제를 조절하기 위하여 배치체계에서 내적으로 쓰인다. 또한 그것은 응용프로그람작성자들이 사용할수 있다.

두개의 다른 클라스들을 포함한다. QLayoutIterator는 응용프로그람작성자들이 볼수 있는 클라스로서 명시적으로 공유된다. QLayoutIterator는 모든 작업을 수행하는 QGLayoutIterator를 포함한다. 배치관리자클라스를 반복하는 방법을 알고있는 QGLayoutIterator의 파생클라스를 창조해야 한다.

이 경우에는 간단한 실현을 선택한다. 즉 목록에서의 옹근수첨수와 목록의 지적자를 보관한다. QGLayoutIterator의 매개 파생클라스는 구성자와 함께 current()와 next(), takeCurrent()를 실현해야 한다. 실례에서는 해체자를 요구하지

않는다.

```
class CardLayoutIterator : public QGLayoutIterator
{
public:
  CardLayoutIterator( QPtrList<QLayoutItem> *l ) : idx( 0 ), list( l ) {}
  QLayoutItem *current() { return idx < int(list->count()) ? list->at(idx) : 0;  }
  QLayoutItem *next() { idx++; return current(); }
  QLayoutItem *takeCurrent() { return list->take( idx ); }

private:
  int idx;
  QPtrList<QLayoutItem> *list;
};
```

이 배치관리자에 대하여 QLayoutIterator를 돌려주도록 QLayout:iterator()를 실현해야 한다.

```
QLayoutIterator CardLayout::iterator()
{
  return QLayoutIterator( new CardLayoutIterator(&list) );
}
```

addItem()은 배치관리자항목들에 대한 기정배치전략을 실현한다. 이것은 QLayout::add()에 의하여, 배치관리자를 부모로 가지는 QLayout구성자에 의하여 사용되며 자동추가기능을 실현하는데 쓰인다. 자기의 배치관리자가 파라메터들을 요구하는 고급한 배치선택을 가지고있으면 QGridLayout::addMultiCell()와 같은 여분의 호출함수를 제공해야 한다.

```
void CardLayout::addItem( QLayoutItem *item )
{
  list.append( item );
}
```

배치관리자는 추가된 항목들의 응답능력을 물려받는다. QLayoutItem가 QObject를 계승하지 않으므로 항목들을 수동적으로 삭제해야 한다. 함수 QLayout::deleteAllItems()는 배치관리자안의 모든 항목들을 삭제하기 위하여 우에서 정의한 복사구성자를 사용한다.

```
CardLayout::~CardLayout()
{
  deleteAllItems();
}
```

setGeometry()함수는 실제로 배치를 수행한다. 인수로서 주어진 직4각형은 margin()을 포함하지 않는다. 적절하다면 항목들사이의 거리로서 spacing()을 사용한다.

```
void CardLayout::setGeometry( const QRect &rect )
{
  QLayout::setGeometry( rect );

  QPtrListIterator<QLayoutItem> it( list );
  if (it.count() == 0)
    return;

  QLayoutItem *item;

  int i = 0;

  int w = rect.width() - ( list.count() - 1 ) * spacing();
```

```
  int h = rect.height() - ( list.count() - 1 ) * spacing();

  while ( (item = it.current()) != 0 ) {
    ++it;
    QRect geom( rect.x() + i * spacing(), rect.y() + i * spacing(), w, h );
    item->setGeometry( geom );
    ++i;
  }
}
```

sizeHint()와 minimumSize()는 보통 실현에서 서로 비슷하다. 두 함수가 돌려준 크기들은 margin()이 아니라 spacing()을 포함해야 한다.

```
QSize CardLayout::sizeHint() const
{
  QSize s( 0, 0 );
  int n = list.count();
  if ( n > 0 )
    s = QSize( 100, 70 ); // start with a nice default size
  QPtrListIterator<QLayoutItem> it( list );
  QLayoutItem *item;
  while ( (item = it.current()) != 0 ) {
    ++it;
    s = s.expandedTo( item->minimumSize() );
  }
  return s + n * QSize( spacing(), spacing() );
}

QSize CardLayout::minimumSize() const
{
  QSize s( 0, 0 );
  int n = list.count();
  QPtrListIterator<QLayoutItem> it( list );
  QLayoutItem *item;
  while ( (item = it.current()) != 0 ) {
    ++it;
    s = s.expandedTo( item->minimumSize() );
  }
  return s + n * QSize( spacing(), spacing() );
}
```

(3) 알아두어야 할것

이 배치관리자는 heightForWidth()를 실현하지 않는다.

우리는 QLayoutItem::isEmpty()를 무시한다. 이것은 은페된 창문부품들을 볼수 있는것으로 취급한다는것을 의미한다.

복잡한 배치에서 속도는 계산값들을 고속완충함으로써 크게 증가될수 있다. 그 경우에 QLayoutItem::invalidate()를 실현하여 고속완충된 자료를 불결한것으로서 표식한다.

QLayoutItem::sizeHint() 등의 호출은 비용이 들수 있으므로 같은 함수에서 후에 다시 요구되는 값은 국부변수에 보관하여야 한다.

같은 함수에서 같은 항목에 대하여 QLayoutItem::setGeometry()를 두번 호출하지 말아야 한다. 항목이 여러개의 자식창문부품을 가지면 매번 완전배치를 수행해야 하므로 거기에 비용이 매우 많이 든다. 그대신에 기하학적도형을 계산하여 설정한다. (이것은 배치관리자에만 제한되지 않고 자체의 resizeEvent()를 실현하는 경우에도 같은 조작을 수행해야 한다.)

제3절. 자리표계

Qt의 그리기장치는 그림을 그릴수 있는 2차원평면이다. QWidget와 QPixmap, QPicture, QPrinter는 모두 그리기장치이다. QPainter는 그러한 장치우에 그릴수 있는 객체이다.

그리기장치의 기정자리표계는 왼쪽웃구석에 자기 원점이 있다. X는 오른쪽으로 증가하고 Y는 아래로 증가한다. 단위는 화소에 기초하는 장치에서 1화소이고 인쇄기에서 1point이다.

1. 실례

그림 5-1은 그리기장치의 왼쪽웃구석을 크게 확대한 부분을 보여준다.

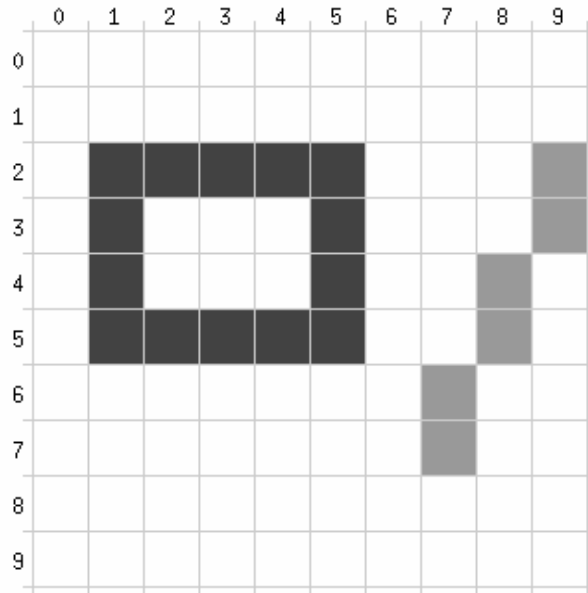



그림 5-1. 자리표계

직4각형과 직선은 다음의 코드에 의하여 (격자와 설명에서 준 색을 추가하여) 그려진다.

```
void MyWidget::paintEvent( QPaintEvent * )
{
  QPainter p( this );
  p.setPen( darkGray );
  p.drawRect( 1,2, 5,4 );
  p.setPen( lightGray );
  p.drawLine( 9,2, 7,7 );
}
```

drawRect()에서 그리는 모든 화소는 지정된 크기(5×4화소)안에 있다. 이것은 일부 도구들에서는 다르다. Qt에서 사용자가 지정하는 크기는 그려진 화소들을 정확히 둘러싼다. 이것은 QPainter의 관련한 모든 함수들에 적용된다.

마찬가지로 drawLine()호출은 직선의 량끝점을 그린다.

표 5-1에 자리표계와 가장 밀접히 련관된 클라스들을 보여준다.

표 5-1. 자리표계관련클라스

QPoint	자리표계에서 하나의 2차원점. Qt에서 점들을 취급하는 대부분의 함수는 하나의 QPoint인수 혹은 2개의 int를 받아들일수 있다. 실례로 QPainter::drawPoint().
QSize	하나의 2차원벡토르. 내부적으로 QPoint와 QSize는 같지만 점과 크기는 다르므로 두개의 클라스가 존재한다. 또한 대다수 함수들은 하나의 QSize 혹은 2개의 int를 받아들인다. 실례로 QWidget::resize().
QRect	2차원직4각형. 대부분의 함수들은 하나의 QRect 혹은 4개의 int를 받아들인다. 실례로 QWidget::setGeometry().
QRegion	임의의 점들의 모임으로서 모든 표준연산모임 실례로 QRegion::intersect() 또한 공용체나 령역(region)과 같은 직4각형들의 목록을 돌려주는 적은 량의 일반함수를 포함한다. QRegion은 례하면 QPainter::setClipRegion()와 QWidget::repaint(), QPaintEvent::region()에 의하여 사용된다.
QPainter	그리기하는 클라스. 같은 코드로 어떤 장치에 그릴수 있다. 장치들사이에 차이가 있다. QPrinter::newPage()가 좋은 실례이지만 QPainter는 모든 장치에서 같은 방식으로 작업한다.
QPaintDevice	QPainter가 그릴수 있는 장치. 두개의 내부장치(둘다 화소에 기초)와 두개의 외부장치 QPrinter와 QPicture(QPainter지령들을 파일이나 다른 QIODevice에 기록하고 그것들을 되살린다.)가 있다. 다른 장치도 정의할수 있다.

2. 변환

Qt의 기정자리표계가 우에 서술한것처럼 작업한다하더라도 QPainter는 또한 임의의 변환도 유지한다.

첫 단계에서는 세계(world)변환행렬을 사용하여 자기 모형에서 객체들의 방향을 맞추고 위치를 지정한다. Qt는 이 행렬에 대하여 조작하도록 하기 위하여 QPainter::rotate(), QPainter::scale(), QPainter::translate() 등 메쏘드들을 제공한다.

QPainter::save()와 QPainter::restore()는 이 행렬을 보관하고 되살린다. 또한 QWMatrix객체들과 QPainter::worldMatrix(), QPainter::setWorldMatrix()를 사용하여 이름있는 행렬을 보관하고 리용한다.

둘째 단계에서는 창문을 사용한다. 창문(window)은 모형자리표계의 보기경계를 서술한다. 행렬은 객체들의 위치를 정하고 QPainter::setWindow()는 자리표계에서 보이는 창문의 위치를 정한다.

셋째 단계는 보기구역(viewport)을 사용한다. 보기구역 역시 보기경계를 서술하지만 장치자리표계이다. 보기구역과 창문은 같은 직4각형을 서술하지만 자리표계가 다르다.

화면에서 기정은 사용자가 그리고있는 전체QWidget 혹은 QPixmap이다.

그러므로 그려지는 각 객체는 QPainter::worldMatrix()에 의하여 모형자리표계로 변환된 다음 QPainter::window()와 QPainter::viewport()에 의하여 그리기장치상의 위치가 정해진다.

하나 또는 두개의 단계없이 수행할수도 있다. 실례로 자기 목적이 어떤 비례로 확장하여 그리는것이라면 QPainter::scale()를 리용하는것이 옳다. 자기 목적이 고정크기자리표계를 사용하는것이라면 QPainter::setWindow()가 리상적이다.

여기에 3개의 기구를 모두 사용하는 간단한 실례가 있다. 시계를 그리는 함수는 aclock/aclock.cpp실례에 있다. 더 읽기전에 실례를 콤파일하고 실행할것을 권고한다.

특히 창문을 각이한 크기로 조절해보시오.

```
void AnalogClock::drawClock( QPainter *paint )
{
  paint->save();
```

우선 인쇄기의 상태를 보관하여 호출하는 함수가 우리가 사용하려는 변환에 의하여 혼돈하지 않도록 담보할수 있다.

```
paint->setWindow( -500,-500, 1000,1000 );
```

모형자리표계를 1000×1000창문으로 설정한다. 여기서 중심은 0,0이다.

```
QRect v = paint->viewport();
int d = QMIN( v.width(), v.height() );
```

우리가 시계를 요구하고 장치는 바른4각형이 아니므로 현재의 보기구역을 찾고 그 최소의 변을 계산한다.

```
paint->setViewport( v.left() + (v.width()-d)/2, v.top() + (v.height()-
d)/2, d, d );
```

그다음 낡은것의 중심에 있는 새로운 바른4각형 보기구역을 설정한다.

이제는 보기에서 수행한다. 현시점에서 0,0주위의 1000×1000구역에서 그릴 때 우리가 그리는것은 출력장치에 알맞는 최대로 가능한 바른4각형안에 표시된다.

그리기를 시작한다.

```
QPointArray pts;
```

pts는 점들을 보관하는 일시변수이다.

다음에 3개의 그리기블로크, 하나는 시침, 다른 하나는 분침, 끝으로 문자판을 그린다. 우선 시침을 그린다.

```
paint->save();
paint->rotate( 30*(time.hour()%12-3) + time.minute()/2 );
```

그리기장치(painter)를 보관하고 그것을 회전하여 한개 축이 시침을 가리키게 한다.

```
pts.setPoints( 4, -20,0,  0,-20, 300,0, 0,20 );
paint->drawConvexPolygon( pts );
```

pts를 4각형으로 설정하여 시침이 3시를 가리키도록 그린다. 회전에 의하여 그려지는것은 오른쪽방향을 가리킨다.

```
paint->restore();
```

보관된 그리기장치를 복귀하여 회전을 취소시킨다. 또한 rotate(-30)를 호출할수 있지만 둥그리기오유가 발생할수 있으므로 save()와 restore()를 사용하는것이 좋다. 다음에 거의 같은 방법으로 분침을 그린다.

```
paint->save();
paint->rotate( (time.minute()-15)*6 );
pts.setPoints( 4, -10,0, 0,-10, 400,0, 0,10 );
paint->drawConvexPolygon( pts );
paint->restore();
```

유일한 차이는 회전각을 계산하는 방법과 다각형의 모양이다.

마지막으로 그려야 할것은 시계의 문자판이다.

```
for ( int i=0; i<12; i++ ) {
  paint->drawLine( 440,0, 460,0 );
  paint->rotate( 30 );
}
```

12개의 짧은 시간선이 30도간격으로 되여있다. 그 끝에서 그리기객체가 아주 비효과적인 방법으로 회전되지만 그리지 않으므로 문제가 생기지 않는다.

```
    paint->restore();
  }
```

함수의 마지막 행은 그리기객체를 되살리므로 호출자는 우리가 수행한 모든 변환의 영향을 받지 않는다.

제4절. 창문기하

QWidget는 창문부품의 기하를 취급하는 여러개의 함수들을 제공한다(표 5-2). 그 일부 함수들은 순수의뢰령역(즉 창문틀을 제외한 창문)에 대하여 조작하며 다른 함수들은 창문틀에 대하여 조작한다. 변이는 가장 일반적인 사용법을 투명하게 포함하는 방법으로 수행된다.

표 5-2. 창문기하함수

창문틀의 포함	x(), y(), frameGeometry(), pos() 그리고 move()
창문틀의 제외	geometry(), width(), height(), rect() 그리고 size()

제일 웃준위의 장식된 창문부품들에 대해서만 차이가 커진다. 모든 자식창문부품들에 대하여 틀(frame)기하는 창문부품의 의뢰기기하와 같다.

그림 5-2는 사용하는 함수들의 대부분을 설명한다.

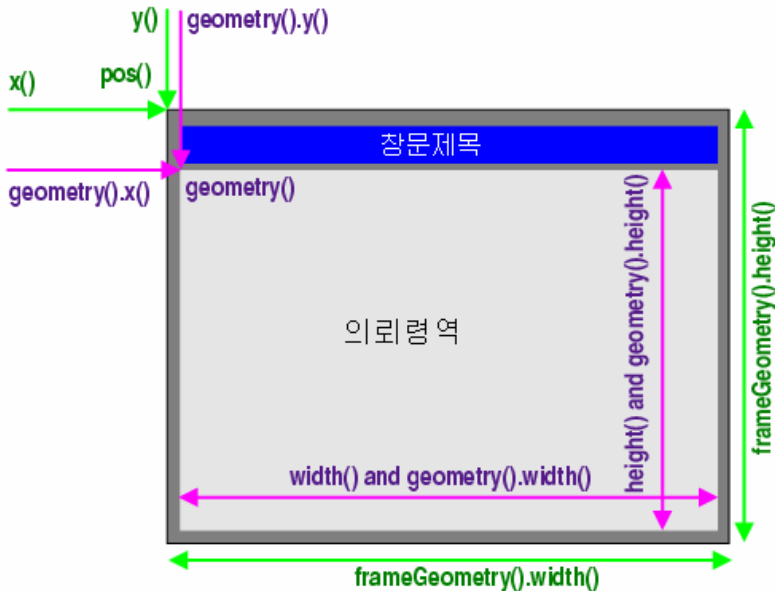



그림 5-2. 창문구조

1. Unix/X11특성

Unix/X11에서 창문은 창문관리기가 창문을 장식할 때까지 틀을 가지지 않는다. 이것은 show()를 호출하여 창문이 받아들이는 첫 그리기사건후에 바로 어느한 시점에서 비동기적으로 발생한다. 혹은 전혀 발생하지 않는다. X11가 유연하므로 창문이 얻는 장식들에 대한 안전한 가설을 만들수 없다.

더우기 개발도구는 화면우에 창문들을 단순히 배치할수 없다. Qt가 해야 할 일은 창문관리기에 일정한 암시를 보내는것이다. 개별적인 프로세스인 창문관리기는 그 암시를 따르거나 무시하거나 오해할수 있다. 부분적으로 명백하지 않은 ICCCM (Inter-Client Communication Conventions Manual)으로 인하여 창문배치는 현존창문관리

기들에서 완전히 다르게 처리된다.

X11은 창문이 장식되면 틀기하를 얻는 간단한 방법을 제공한다. Qt는 오늘날 존재하는 넓은 범위의 창문관리기들에서 작업하는 계발규칙과 코드에 의해 이 문제를 해결한다.

X11은 창문을 최대화하는 방법을 제공하지 않는다. 그러므로 Qt의 showMaximized()함수는 그 특성을 모의해야 한다. 결과는 frameGeometry()의 결과와 창문관리기가 적당한 창문배치를 수행하는 능력에 의존하며 그 어느것도 담보될수 없다.

2. 창문기하의 복귀

현대응용프로그람들의 공통과제는 후의 쎄숀에서 창문기하를 되살리는것이다. Windows에서는 geometry()의 결과를 기본적으로 보관하고 다음 쎄숀에서 show()를 호출하기전에 setGeometry()를 호출한다. X11에서는 보이지 않는 창문이 아직 틀을 가지지 않으므로 작업하지 않는다. 창문관리기는 후에 창문을 장식하군 한다. 이런 일이 발생할 때 창문은 장식틀의 크기에 따라 화면의 오른쪽아래구석으로 옮긴다. X는 리론적으로 이러한 옮기기를 피하는 방법을 제공한다. 그럼에도 불구하고 거의 모든 창문관리기들이 이 기능을 실현하는데서 실패한다.

그 대책은 show()후에 setGeometry()를 호출하는것이다. 이것은 두가지 결함이 있다. 즉 창문부품이 1ms동안 잘못된 위치에 나타나고(섬광이 생긴다) 현재는 오직 매개의 둘째 창문관리기가 그것을 바로 잡는다. 더 안전한 해결책은 pos()와 size()를 둘다 보관하고 show()를 호출하기전에 resize()와 move()를 리용하여 기하를 회복하는것이다. 그것을 다음의 실례에서 보여준다.

```
MyWidget* widget = new MyWidget
...
QPoint p = widget->pos();   // 위치보관
QSize s = widget->size();   // 크기보관
...
widget = new MyWidget;
widget->resize( s );     // 크기회복
widget->move( p );      // 위치회복
widget->show();        // 창문부품표시
```

이 방법은 MS-Windows와 대부분의 현존 X11창문관리기들에서 작업한다.

제6장. 모듈

Qt를 설치할 때 일정한 모듈들이 서고에 구축된다. Qt Professional판에서는 기본모듈 즉 도구, 핵심, 창문부품, 대화칸, 그림기호보기 그리고 작업공간모듈을 사용할수 있다. 현재 Trolltech는 개별적으로 판매하기 위한 모듈을 제공하지 않는다.

각 판과 련관된 사용허가하에서만 모든 모듈을 사용할수 있다.

제1절. 캔버스모듈

캔버스(그림천)모듈은 QCanvas라고 부르는 고도로 최적화된 2차원도형처리령역을 제공한다. 캔버스는 임의의 개수의 QCanvasItem을 포함할수 있다. 캔버스항목들은 임의의 모양과 크기, 내용을 가질수 있으며 캔버스안에서 자유로 이동할수 있으며 충돌을 검사할수 있다. 캔버스항목들은 자동적으로 캔버스를 질러가도록 설정될수 있으며 동화캔버스항목들은 QCanvasSprite에 의해 유지된다.(3차원도형처리를 요구한다면 Qt의 OpenGL모듈을 참고.)

캔버스모듈은 문서-보기모형을 사용한다. QCanvasView클라스는 캔버스의 특수한 보기를 표시하는데 리용된다. 다중보기는 같은 캔버스에 대하여 동시에 조작할수 있다. 매개 보기는 확대와 같은 기능을 간단히 실현하게 하는 임의의 전송행렬을 캔버스에서 사용할수 있다.

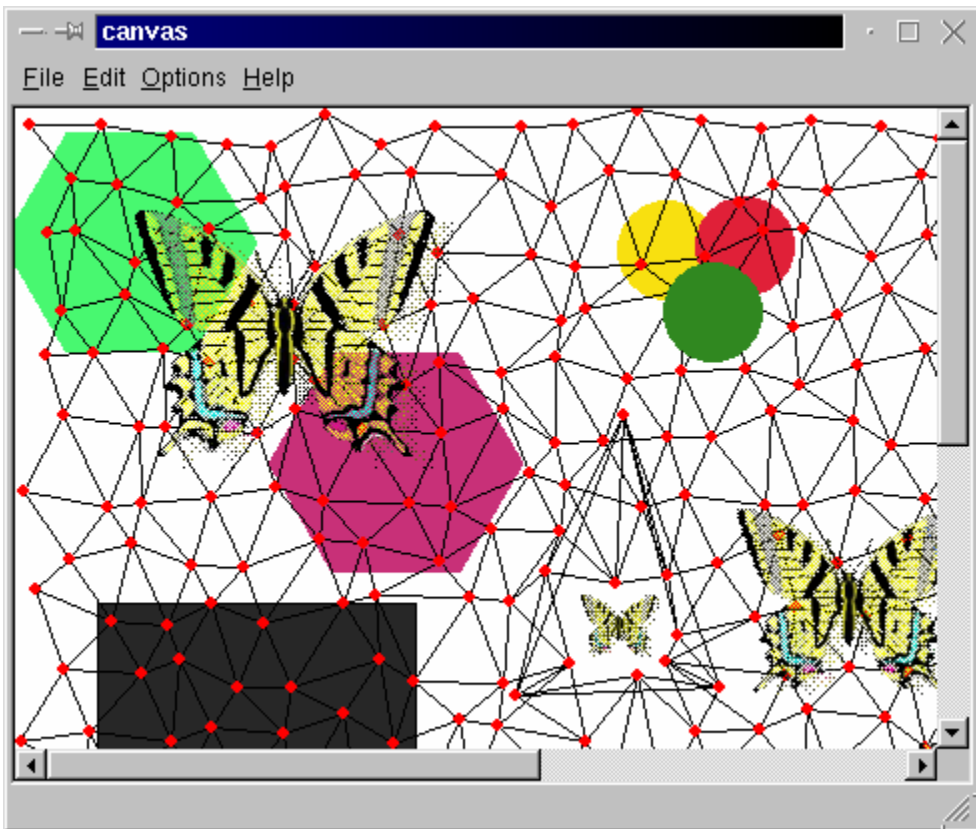



그림 6-1. 캔버스모듈

Qt는 다음과 같이 미리 정의된 QCanvas항목들을 많이 제공한다.

• QCanvasItem -모든 캔버스항목들의 추상기초클라스.
• QCanvasEllipse - 타원 혹은 부채형.
• QCanvasLine - 선분.
• QCanvasPolygon - 다각형.
• QCanvasPolygonalItem - 비직4각형모양을 가지는 항목들의 기초클라스. 대부분의 캔버스항목들은 이 클라스에서 파생된다.
• QCanvasRectangle - 직4각형. 직4각형은 타일모양으로 하거나 또는 회전시킬수 없다. 회전된 직4각형은 QCanvasPolygon을 리용하여 그린다.
• QCanvasSpline - 다중베쎌 스프라인.
• QCanvasSprite - 동화픽스매프.
• QCanvasText - 본문문자렬.

두개 클라스 QCanvasPixmap와 QCanvasPixmapArray는 QCanvasSprite에 의해 사용되며 캔버스에 살아서 움직이는 픽스매프를 표시한다.

더 전문화된 항목들을 캔버스항목클라스들중의 하나로부터 계승에 의해 창조할수 있다. 직접 QCanvasItem를 계승하기보다 QCanvasItem의 파생클라스들중의 하나(보통 QCanvasPolygonalItem)로부터 계승하는것이 가장 간단하다.

(QCanvas능력을 보여주는 실례로서 examples/canvas를 참고하시오.)

제2절. IconView모듈

그림기호보기모듈은 QIconView라고 부르는 강력한 시각화창문부품을 제공한다. API와 기능에서 QIconView는 QListView, QListBox와 밀접히 련관되여있다. 그것은 사용자가 선택, 끌기, 이름변경, 삭제 등을 할수 있는 표식이 있는 픽스매프항목들을 마음대로 포함한다.

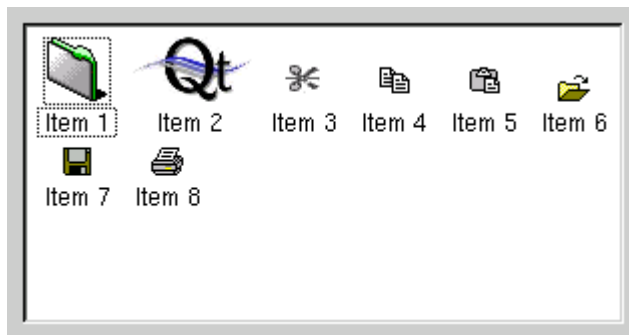



그림 6-2. 그림기호보기모듈

제3절. 망모듈

1. 소개

망모듈은 망프로그람을 쉽게 작성하고 이식할수 있게 하기 위한 클라스들을 제공한다. 본질적으로 세 부류의 클라스들이 있는데 첫째는 QSocket, QServerSocket, QDns 등과 같은 저수준클라스들로서 TCP/IP소케트를 가지고 이식가능한 방식으로 작업할수 있게 한다. 또한 Qt기초서고에는 망규약실현을 위한 추상층을 제공하는 QNetworkProtocol, QNetworkOperation과 같은 클라스들과 이러한 망규약상에서 조작하는 QUrlOperator가 있다. 끝으로 셋째 부류의 망클라스들은 이전의 클라스들로서 특히 URL문법해석과 그와 류사한 일을 하는 QUrl과 QUrlInfo이 있다.

첫째 부류의 클라스들(QSocket, QServerSocket, QDns, QFtp, 등)은 Qt의 《망》모듈에 포함된다.

QSocket클라스들은 QNetwork클라스들과 직접 련관되지 않지만 QSocket는 직접 련관되여있으며 QNetwork클라스들과 직접 련관되는 망규약을 실현하는데 사용된다. 실례로 QFtp클라스(FTP규약을 실현)는 QSocket를 사용한다. 그러나 QSocket를 통신규약실현에 사용할 필요가 없다. 실례로 QLocalFs(망통신규약으로서 국부파일체계의 실현)는 QDir를 사용하며 QSocket를 사용하지 않는다. QNetworkProtocol을 사용하여 계층구조에 적합한 모든것을 실현할수 있고 URL을 사용하여 호출할수 있다. 이것은 실례로 직렬련결을 리용하여 수자식카메라로부터 그림을 읽어들일수 있는 통신규약이다.

2. QUrlOperator와 QNetworkOperation을 리용하여 독립적인 망통신규약작성

현존망통신규약실현을 사용하는것과 URL에 대하여 조작하는것은 아주 간단하다. 실례로 FTP봉사기로부터 국부파일체계에로 파일의 내리적재는 다음의 코드로 수행할수 있다.

```
QUrlOperator op;
op.copy( "ftp://ftp.trolltech.com/qt/source/qt-2.1.0.tar.gz", "file:/tmp", FALSE );
```

물론 FTP통신규약의 실현은 그것을 수행하는데 사용할수 있고 등록되여야 한다.

또한 등록부창조, 파일삭제, 이름변경과 같은것들을 수행할수 있다. 실례로 자기의 FTP등록자리(account)에서 폴더를 다음과 같이 창조한다.

```
QUrlOperator op( "ftp://username:password@host.domain.no/home/username" );
op.mkdir( "New Directory" );
```

(모든 유효조작은 QUrlOperator클라스문서를 참고하시오.)

망작업은 비동기적으로 진행되므로 어떤 조작을 요구하는 함수호출은 보통 그 조작이 끝나기전에 되돌아온다. 이것은 그 함수가 실패나 성공을 가리키는 값을 돌려줄수 없다는것을 의미한다. 대신에 돌림값은 항상 QNetworkOperation의 지적자이며 이 객체는 그 조작에 대한 모든 정보를 보관한다.

실례로 QNetworkOperation은 이 조작의 상태를 돌려주는 메쏘드를 가진다. 이것을 리용하여 임의의 순간에 그 조작의 상태를 알아볼수 있다. 또한 그 객체는 QUrlOperator메쏘드에 넘어온 인수들과 조작의 형, 기타 정보를 사용할수 있게 한다(QNetworkOperation클라스문서를 참고).

QUrlOperator는 조작의 진행정형을 알리고 신호를 발생한다. QUrlOperator의 URL에 대하여 조작하는 많은 메쏘드들을 호출할수 있으므로 그 조작들은 모두 대기렬에 넣는다. 그러므로 QUrlOperator가 방금 어떤 조작을 처리했는지 알수 없다. 어느 조작이 발생되였는가를 꼭 알아야 하므로 매개 신호의 마지막 인수는 방금 처리되여 신호를 발생한 QNetworkOperation객체의 지적자로 한다.

이러한 조작의 일부는 start()신호를 보내며 (만일 이것이 의미를 가진다면) 그 일부는 처리과정에 신호들을 보낸다. 모든 조작은 그것이 수행된 후에 finished()신호를 보낸다. finished()신호와 함께 얻어지는 QNetworkOperation지적자를 사용하여 어떤 조작이 성공적으로 끝났는가를 알아볼수 있다. QNetworkOperation::state()가 QNetworkProtocol::StDone와 같으면 조작이 성공적으로 끝난것이고 그것이 QNetworkProtocol::StFailed이면 조작은 실패한것이다.

실례: QUrlOperator::finished(QNetworkOperation *) 에 련결할수 있는 처리부

```
void MyClass::slotOperationFinished( QNetworkOperation *op )
```

```
{
  switch ( op->operation() ) {
  case QNetworkProtocol::OpMkDir:
    if ( op->state() == QNetworkProtocol::StFailed )
      qDebug( "Couldn't create directory %s", op->arg( 0 ).latin1() );
    else
      qDebug( "Successfully created directory %s", op->arg( 0 ).latin1() );
    break;
  // ... and so on
  }
}
```

앞에서 언급한것처럼 일부 조작은 다른 신호들도 보낸다. 실례로 자식목록조작을 고찰하자. (FTP봉사기에서 등록부를 읽어들인다.)

```
QUrlOperator op;

MyClass::MyClass() : QObject(), op( "ftp://ftp.trolltech.com" )
{
  connect( &op, SIGNAL(
       newChildren( const QValueList<QUrlInfo> &, QNetworkOperation * ) ),
                         this,SLOT(slotInsertEntries(const
                         QValueList<QUrlInfo>&,QNetworkOperation *)));
  connect( &op, SIGNAL( start( QNetworkOperation * ) ),
       this, SLOT( slotStart( QNetworkOperation *) ) );
  connect( &op, SIGNAL( finished( QNetworkOperation * ) ),
       this, SLOT( slotFinished( QNetworkOperation *) ) );
}

void MyClass::slotInsertEntries(const QValueList<QUrlInfo>&info,QNetworkOperation*)
{
  QValueList<QUrlInfo>::ConstIterator it = info.begin();
  for ( ; it != info.end(); ++it ) {
    const QUrlInfo &inf = *it;
    qDebug( "Name: %s, Size: %d, Last Modified: %s",
      inf.name().latin1(), inf.size(), inf.lastModified().toString().latin1() );
  }
}

void MyClass::slotStart( QNetworkOperation * )
{
  qDebug( "Start reading '%s'", op.toString().latin1() );
}

void MyClass::slotFinished( QNetworkOperation *operation )
{
```

```
if ( operation->operation() == QNetworkProtocol::OpListChildren ) {
  if ( operation->state() == QNetworkProtocol::StFailed )
    qDebug( "Couldn't read '%s'! Following error occurred: %s",
      op.toString().latin1(), operation->protocolDetail().latin1() );
  else
    qDebug( "Finished reading '%s'!", op.toString().latin1() );
}
}
```

이 실례들은 QUrlOperator와 QNetworkOperation의 사용법을 보여준다. 또한 망확장은 쓸모있는 실례코드를 포함한다.

1) 자체의 망통신규약의 실현

QNetworkProtocol은 망통신규약을 실현하기 위한 기초클라스와 망통신규약의 동적인 등록과 등록해제를 위한 구성방식을 제공한다. 이 구성방식을 사용하면 비동기적프로그람작성에 대하여 관심을 두지 않아도 된다. 그것은 구성방식에 이것이 은페되여있고 그것이 모든 작업을 수행하기때문이다.

알아두기: 모든 망통신규약들에서 사용할수 있는 망통신규약용 기초클라스를 설계하는것은 힘들다. 여기에 서술한 구성방식은 파일체계와 같은 모든 종류의 계층구조에서 작업하도록 설계된다. 그러므로 계층구조로서 해석할수 있고 URL을 거쳐서 호출할수 있는 모든것은 망통신규약으로 실현될수 있으며 Qt에서 간단히 사용할수 있다. 이것은 파일체계에만 국한되지 않는다.

망통신규약을 실현하려면 QNetworkProtocol로부터 파생된 클라스를 창조한다.

다른 클라스들은 이 망통신규약실현을 사용하여 조작한다. 그러므로 다음의 보호성원들을 재정의하여야 한다.

```
void QNetworkProtocol::operationListChildren( QNetworkOperation *op );
void QNetworkProtocol::operationMkDir( QNetworkOperation *op );
void QNetworkProtocol::operationRemove( QNetworkOperation *op );
void QNetworkProtocol::operationRename( QNetworkOperation *op );
void QNetworkProtocol::operationGet( QNetworkOperation *op );
void QNetworkProtocol::operationPut( QNetworkOperation *op );
```

이 메쏘드들의 재정의에 대한 몇가지 알아두기: 인수로서는 항상 QNetworkOperation의 지적자를 얻는다. 이 지적자는 현재 상태에서 조작에 대한 모든 정보를 보유한다. 그와 같은 조작을 처리하기 시작하면 상태를 QNetworkProtocol::StInProgress로 설정한다. 조작에 대한 처리가 끝나면 그것이 성공하였을 때 상태를 QNetworkProtocol::StDone으로 설정하고 오유가 발생하면 QNetworkProtocol::StFailed로 설정한다. 오유가 발생하였으면 오유코드를 설정해야 하며(QNetworkOperation::setErrorCode()참고) 자세한것(실례로 오유통보)을 알려면 이 통보를 조작지적자로 설정할수도 있다.(QNetworkOperation::setProtocolDetail()참고.) 또한 QNetworkOperation지적자로부터 조작에 대한 모든 관련정보(형, 인수 등)를 얻는다. (어떤 인수들을 얻고 설정할수 있는가에 대하여 자세히 알려면 QNetworkOperation의 클라스문서를 참고하시오.)

어떤 조작함수를 재정의한다면 정확한 시간에 정확한 신호를 발생하는것이 아주 중요하다. 일반적으로 조작의 마감에 (성공적으로 조작처리를 끝냈거나 오유가 발생하였을 때) 인수로서 망조작을 가지는 finished()를 항상 발생한다. 전체 망구성방식은 정확히 발생된 finished()신호에 기초한다. 그때 조작들에 고유한 좀 더 전문화된 신호들이 있다.

- operationListChildren에서 발생하는것:

start() : 자식들의 렬거를 시작하기 바로전에
newChildren() : 새 자식들을 읽어들일 때
• operationMkDir에서 발생하는것:
createdDirectory() : 등록부를 창조한 후에
newChild() (혹은 newChildren()) : 등록부를 창조한 후에(새 등록부가 새 자식이므로)
• operationRemove에서 발생하는것:
removed() : 자식이 삭제된 후에
• operationRename에서 발생하는것:
itemChanged() : 자식의 이름이 변경된 후에
• operationGet에서 발생하는것:
data() : 새 자료를 읽어들였을 때마다
dataTransferProgress() : 새 자료를 읽어들였을 때마다 현재 읽어들인 자료량을 보여준다.
• operationPut에서 발생하는것:
dataTransferProgress() : 자료를 써넣었을 때마다 써넣은 자료량을 보여준다. 비록 이 조작이 호출될 때 모든 자료를 알고있어도 한번에 전체 자료를 쓰지 않고 GUI가 봉쇄되는것을 피하기 위하여 한걸음씩 수행할것을 제안한다. 또한 증분적으로 수행하는 수법은 그 진척과정을 사용자가 시각적으로 볼수 있게 한다.

그리고 마감에 항상 finished()신호를 발생한다(QNetworkProtocol클라스문서 참고).

여기에 어떤 QNetworkOperation인수들을 얻을수 있으며 어느 함수에서 어떤 인수들을 설정하여야 하는가 하는 목록이 있다. (자기가 작업하여야 할 URL을 얻으려면 URL연산자의 지적자를 돌려주는 QNetworkProtocol::url()메쏘드를 사용한다. 그것을 사용하여 경로, 주콤퓨터, 이름려과기 등을 얻을수 있다.)
• operationListChildren에서
없다.
• operationMkDir에서
QNetworkOperation::arg(0)은 창조하려는 등록부의 이름을 포함한다.
• operationRemove에서
QNetworkOperation::arg(0)은 삭제하려는 파일의 이름을 포함한다. 보통 이것은 상대적인 이름이다. 그러나 절대적일수도 있다. QUrl(op->arg(0)).fileName()을 사용하여 파일이름을 얻는다.
• operationRename에서
QNetworkOperation::arg(0)은 변경하려는 파일의 이름을 포함한다.
QNetworkOperation::arg(1)은 변경된 이름을 포함한다.
• operationGet에서
QNetworkOperation::arg(0)은 검색하여야 할 파일의 완전URL을 포함한다.
• operationPut에서
QNetworkOperation::arg(0)은 자료가 보관되는 파일의 완전URL을 포함한다.
QNetworkOperation::rawArg(1)은 QNetworkOperation::arg(0)안에 보관될 자료를 포함한다.

요약: 하나의 조작함수를 재정의한다면 여러개의 특수신호들을 발생해야 하며 마지막에는 성공했는가 실패했는가에 관계없이 항상 finished()신호를 발생해야 한다. 또한 처리과정에 QNetworkOperation의 상태를 변경하여야 한다. 또한 조작이 진행될 때 QNetworkOperation인수들을 얻거나 설정할수 있다.

자기가 실현하는 망통신규약이 이 조작들의 부분모임만을 요구하는 경우도 있을수

있다. 그러한 경우에 간단히 통신규약에 의하여 제공되는 조작들을 재정의한다. 또한 자기가 지원하는 조작들을 지정해야 한다. 이것은 재정의에 의하여 달성된다.

```
int QNetworkProtocol::supportedOperations() const;
```

이 메쏘드의 실현에서는 다음과 같은 QNetworkProtocol의 렬거값들(지원된 조작들)의 론리합으로 구성되는 int값을 돌려주어야 한다.

- OpListChildren
- OpMkDir
- OpRemove
- OpRename
- OpGet
- OpPut

실례로 자기의 통신규약이 자식들의 목록화와 그 이름변경을 지원한다면 자기의 supportedOperations()를 다음과 같이 실현하여야 한다.

```
return OpListChildren | OpRename;
```

재정의해야 할 마지막 메쏘드는 다음과 같다.

```
bool QNetworkProtocol::checkConnection( QNetworkOperation *op );
```

여기서 련결이 성공하면 TRUE를 돌려주어야 한다. (이것은 통신규약에 대한 조작이 진행될수 있다는것을 의미한다.) 련결이 실패하면 FALSE를 돌려주고 다시 련결을 열려고 시도한다. 끝까지 련결을 열수 없으면 (실례로 주콤퓨터를 찾지 못하여) finished()신호를 발생하고 QNetworkProtocol::StFailed상태를 여기서 얻어지는 QNetworkOperation지적자로 설정한다.

련결이 성공하면 조작을 하기전에 자체로 검사할 필요가 없다. 망구성방식이 바로 이것을 수행하는데 그것은 checkConnection()을 리용하여 어떤 조작을 수행할수 있는가를 확인하며 수행할수 없으면 일정한 시간동안 그것을 다시 시도하고 련결이 성공하면 조작함수를 호출할뿐이다.

QUrlOperator(그밖에 실례로 QFileDialog에서 망통신규약을 사용하게 하려면 망통신규약실현을 등록하여야 한다. 이것은 다음과 같이 수행할수 있다.

```
QNetworkProtocol::registerNetworkProtocol( "myprot", new
QNetworkProtocolFactory<MyProtocol> );
```

이 경우에 MyProtocol은 여기서 서술한것처럼 실현한 클라스(QNetworkProtocol에서 파생)이며 통신규약의 이름은 "myprot"이다. 그것을 사용하려면 다음과 같이 해야 한다.

```
QUrlOperator op( "myprot://host/path" );
op.listChildren();
```

끝으로 망통신규약실현의 실례로서 QLocalFs의 실현을 볼수 있다. 또한 망확장은 망통신규약의 실례실현을 포함한다.

2) 오유조종

오유조종은 새로운 망통신규약의 실현이나 그 사용(QUrlOperator을 통하여)에서 모두 중요하다.

조작의 처리에서 망조작을 끝낸 후에 QUrlOperator는 finished()신호를 발생한다. 이것은 처리된 QNetworkOperation의 지적자를 인수로 가진다. 조작의 상태가 QNetworkProtocol::StFailed이면 조작은 오유에 대한 정보를 더 포함한다. 표 6-1의 오유코드들은 QNetworkProtocol 에서 정의되여있다.

표 6-1. QNetworkProtocol 에서 정의된 오유코드

오유	의미
QNetworkProtocol::NoError	오유가 발생하지 않았다.
QNetworkProtocol::ErrValid	조작하고있는 URL이 무효이다.
QNetworkProtocol::ErrUnknownProtocol	조작중에 있는 URL의 통신규약에 사용할수 있는 통신규약실현이 없다. (실례로 통신규약이 http이고 http실현이 등록되지 않은 경우에)
QNetworkProtocol::ErrUnsupported	조작이 통신규약에 의해 지원되지 않는다.
QNetworkProtocol::ErrParse	URL의 오유를 해석한다.
QNetworkProtocol::ErrLoginIncorrect	가입(login)이 요구되지만 사용자이름이나 암호가 틀린다.
QNetworkProtocol::ErrHostNotFound	URL에서 지정된 주콤퓨터를 찾을수 없다.
QNetworkProtocol::ErrListChildren	자식들을 목록할 때 오유가 발생하였다.
QNetworkProtocol::ErrMkDir	등록부를 창조하는동안 오유가 발생하였다.
QNetworkProtocol::ErrRemove	자식을 삭제하는데 오유가 발생하였다.
QNetworkProtocol::ErrRename	자식의 이름을 변경할 때 오유가 발생하였다.
QNetworkProtocol::ErrGet	자료를 얻을 때(검색할 때) 오유가 발생하였다.
QNetworkProtocol::ErrPut	자료를 넣을 때(올리적재할 때) 오유가 발생하였다.
QNetworkProtocol::ErrFileNotExisting	조작에 의해 요구되는 파일이 존재하지 않는다.
QNetworkProtocol::ErrPermissionDenied	조작수행에 대한 허가가 금지되였다.

QNetworkOperation::errorCode()는 이 코드들중 하나를 돌려준다. 오유코드를 추가적으로 정의하는 자체의 망통신규약실현을 사용한다면 다른것을 돌려준다.

또한 QNetworkOperation::protocolDetails()는 사용자에게 현시하는데 적합한 오유통보를 포함하는 문자렬을 돌려줄수도 있다.

자체의 망통신규약을 실현한다면 발생한 오유를 통보하여야 한다. 우선 그 순간에 처리중에 있는 QNetworkOperation에 늘 접근할수 있어야 한다. 이것은 현재 망조작의 지적자를 돌려주거나 그 순간에 아무런 조작도 처리되지 않는다면 0을 돌려주는 QNetworkOperation::operationInProgress()에 의해 수행된다.

오유가 발생하고 오유를 조종할 필요가 있다면 다음과 같이 한다.

```
if ( operationInProgress() ) {
    operationInProgress()->setErrorCode( error_code_of_your_error );
    operationInProgress()->setProtocolDetails( detail ); // optional
    emit finished( operationInProgress() );
    return;
```

}

QUrlOperator에 대한 련결 등은 자동적으로 수행된다. 또한 현재상태에서 수행할 수 있는 조작이 더는 없으므로 실지 오유가 치명적인것이라면(실례로 주콤퓨터를 찾을수 없다면) finished()를 발생하기전에 QNetworkProtocol::clearOperationStack()를 호출한다.

리상적으로 QNetworkProtocol의 미리 정의된 오유코드들중의 하나를 사용해야 한다. 이것이 불가능하면 자체의 오유코드를 추가할수 있는데 그것들은 보통의 int값들이다. 오유코드의 값이 현존오유코드와 충돌하지 않도록 주의하여야 한다.

qt/examples/network/ftpclient에 실례가 있다. 이것은 상당히 완성된 FTP의뢰기의 실현으로서 파일의 내리적재와 올리적재, 등록부만들기 등을 지원하며 QUrlOperator에 의하여 모든것이 수행된다.

또한 QFtp(qt/src/network/qftp.cpp에서) 혹은 qt/examples/network/networkprotocol/ nntp.cpp의 실례를 고찰할수도 있다.

제4절. Qt OpenGL 3차원도형처리

OpenGL은 3차원도형을 그리기 위한 표준API이다.

OpenGL은 오직 3차원그리기만을 처리하며 GUI프로그람작성문제는 거의 제공하지 않는다. OpenGL응용프로그람의 사용자대면부는 X가동환경에서 Motif, Windows에서 MFC, 혹은 두 가동환경에서 Qt와 같은 도구묶음에 의해 창조되여야 한다.

Qt OpenGL모듈은 Qt응용프로그람에서 OpenGL을 간단히 사용할수 있게 한다. 이것은 다른 Qt창문부품들처럼 사용할수 있는 OpenGL창문부품클라스들을 제공한다. 그러나 그것은 OpenGL API에 의하여 내용을 그릴수 있는 OpenGL현시완충기를 열수 없다.

Qt OpenGL모듈은 가동환경에 의존하는 GLX, WGL 혹은 AGL C API들에서 가동환경에 의존하지 않는 Qt/C++래퍼로서 실현된다. 제공된 기능은 Mark Kilgard의 GLUT서고와 아주 비슷하지만 OpenGL에 훨씬 더 고유하지 않은 보다 많은 GUI기능 즉 완전한 Qt API를 가진다.

1. 설치

X11에 Qt를 설치할 때 configure스크립트는 OpenGL머리부와 서고들이 자기 체계에 설치되는가 자동탐색하며 그렇다면 Qt서고안에 있는 Qt OpenGL모듈을 포함한다. (자기의 OpenGL머리부나 서고들이 비표준등록부에 배치된다면 자기 체계의 config파일안에 있는 QMAKE_INCDIR_OPENGL이나 QMAKE_LIBDIR_OPENGL을 변경할 필요가 있다). 일부 환경설정은OpenGL에 스레드화가 허용될것을 요구하므로 OpenGL이 탐지되지 않으면 configure -thread를 실행한다.

Windows에 Qt를 설치할 때 Qt OpenGL모듈이 늘 포함된다.

Qt OpenGL모듈은 Qt Professional판에서 사용하는것이 허용되지 않는다. OpenGL지원을 요구한다면 Qt Enterprise판으로의 갱신을 고려해야 한다.

X11에서 Mesa의 사용에 대한 알아두기: 3.1판이전의 Mesa는 서고들에 GL과 GLU대신에 MesaGL와 MesaGLU라는 이름을 사용한다. 3.1이전판의 Mesa를 사용하려면 이러한 서고이름을 사용하도록 Makefile들을 변경해야 한다. 가장 간단한 방법은 사용하고있는 config파일안의 QMAKE_LIBS_OPENGL행을 편집하는것이며 즉 "-lGL -lGLU"를 "-lMesaGL -lMesaGLU"로 변경한 다음 configure를 다시 실행하는것이다.

2. QGL클라스들

Qt에서 OpenGL지원클라스들은 다음과 같다.

- QGLWidget: OpenGL장면들을 그리는데 사용하기 쉬운 Qt창문부품.
- QGLContext: OpenGL그리기상황을 은페한다.
- QGLFormat: 그리기상황의 현시형식을 지정한다.
- QGLColormap: GL첨수방식에서 색인화된 색략도를 조종한다.

수많은 응용프로그람들은 높은수준의 QGLWidget클라스만 요구한다. 다른 QGL클라스들은 고급한 기능을 제공한다.

QGL문서는 OpenGL프로그람작성법을 잘 알고있는것을 전제로 한다. 그것이 처음이라면 http://www.opengl.org/이 출발점으로 된다.

제5절. SQL모듈

Qt의 SQL클라스들은 다음과 같다.

QSql, QSqlCursor, QSqlDatabase, QSqlDriver, QSqlDriverPlugin, QSqlEditorFactory, QSqlError, QSqlField, QSqlFieldInfo, QSqlForm, QSqlIndex, QSqlPropertyMap, QSqlQuery, QSqlRecord, QSqlRecordInfo, QSqlResult, QSqlSelectCursor

이상의 SQL클라스들은 Qt응용프로그람들에 대한 원만한 자료기지통합을 제공한다.

이 절에서는 적어도 SQL에 대한 기초지식을 가지고 있다고 가정한다. 간단한 SELECT, INSERT, UPDATE, DELETE지령들을 리해하고 있어야 한다. 비록 QSqlCursor클라스가 SQL에 대한 지식을 요구하지 않는 자료기지열람과 편집을 위한 대면부를 제공하여도 SQL에 대한 기초적인 리해가 상당히 요구된다. (SQL자료기지를 설명하는 표준원서는 참고문헌 [2]이다.)

이 모듈개괄은 순수 프로그람적견지에서 클라스를 제시하며 한편 《Qt프로그람개발도구》에서 1장 7절은 창문부품들사이의 주-세부관계의 설정, 구멍내기처리, 외부열쇠탐색조종방법과 같은 높은 수준의 수법들을 보여준다.

이 절의 모든 실례는 소절 8에서 정의된 표들을 사용한다.

1. SQL모듈구성방식

SQL클라스들은 3개의 층으로 구성된다. 즉

① 사용자대면부층. 이 클라스들은 자료원천으로 QSqlCursor를 사용함으로써 자료기지안에 있는 표나 보기들에 련결할수 있는 자료인식창문부품들을 제공한다. 말단사용자들은 이 창문부품들과 직접 교제하여 자료를 열람하거나 편집할수 있다. Qt Designer는 SQL클라스들과 완전히 통합될수 있고 자료를 인식하는 폼을 창조하는데 사용될수 있다. 또한 자료인식창문부품들은 C++코드로 직접 프로그람작성할수 있다. 이 층을 유지하는 클라스들은 QSqlEditorFactory, QSqlForm, QSqlPropertyMap, QDataTable, QDataBrowser 및 QDataView를 포함한다.

②SQL API층. 이 클라스들은 자료기지에 대한 접근을 제공한다. 련결은 QSqlDatabase클라스에 의해 이루어진다. 자료기지교제는 QSqlQuery클라스를 사용하여 SQL지령들을 직접 실행하거나SQL지령들을 자동적으로 구성하는 고수준 QSqlCursor클라스에 의하여 이루어진다. QSqlDatabase, QSqlCursor 및 QSqlQuery와 함께 SQL API층은 QSqlError, QSqlField, QSqlFieldInfo, QSqlIndex, QSqlRecord 및 QSqlRecordInfo에 의하여 지원된다.

③구동프로그람층. 이것은 3개의 클라스 QSqlResult, QSqlDriver 및 QSqlDriverFactoryInterface로 구성된다. 이 층은 자료기지와 SQL클라스들사이의 저수준다리를 제공한다. 이 층은 구동프로그람작성자와만 관련되여있고 표준자료기지응

용프로그람작성에는 거의 사용되지 않으므로 따로따로 문서화된다.

2. SQL구동프로그람플라그인들

Qt SQL모듈은 플라그인들을 사용하여 실행시에 새 구동프로그람들을 동적으로 적재할수 있다.

SQL구동프로그람문서는 특정한 자료기지관리체계를 위한 플라그인구축방법을 서술한다.

플라그인이 일단 구축되면 Qt는 자동적으로 그것을 적재하여 구동프로그람을 QSqlDatabase에 의하여 사용할수 있다(QSqlDatabase::drivers()참고).

3. 자료기지와 련결

QSqlQuery 혹은 QSqlCursor클라스들을 사용하기전에 적어도 하나의 자료기지련결이 창조되여 열려져있어야 한다.

응용프로그람이 하나의 자료기지련결만 요구한다면 QSqlDatabase클라스는 모든 SQL조작들에 대하여 기정적으로 사용되는 련결을 창조할수 있다. 만일 다중자료기지련결이 요구된다면 이것을 쉽게 설정할수 있다.

QSqlDatabase은 qsqldatabase.h머리부파일을 요구한다.

1) 단일자료기지와 련결

자료기지련결의 창조는 간단한 3단계의 과정 즉 구동프로그람을 능동상태로 만들기, 련결정보의 설정, 그리고 련결의 열기로 이루어진다.

```
// sql/overview/connect1/main.cpp로부터
  #include <qapplication.h>
  #include <qsqldatabase.h>
  #include "../connection.h"

  int main( int argc, char *argv[] )
  {
    QApplication app( argc, argv, FALSE );

    QSqlDatabase                    *defaultDB                    =
QSqlDatabase::addDatabase( DB_SALES_DRIVER );
    defaultDB->setDatabaseName( DB_SALES_DBNAME );
    defaultDB->setUserName( DB_SALES_USER );
    defaultDB->setPassword( DB_SALES_PASSWD );
    defaultDB->setHostName( DB_SALES_HOST );

    if ( defaultDB->open() ) {
      //  Database  successfully  opened;  we  can  now  issue  SQL
commands.
    }

    return 0;
  }
```

첫째로, QSqlDatabase::addDatabase()를 호출하여 구동프로그람을 능동상태로 만들고 이 련결에 사용하려는 구동프로그람의 이름을 넘긴다. 사용할수 있는 구동프로그람들로서는 QODBC3(Open Database Connectivity, Microsoft SQL Server지원을

포함), QOCI8(Oracle 8과 9), QTDS7(Sybase Adaptive Server), QPSQL7(PostgreSQL 6과 7), QMYSQL3(MySQL), QDB2(IBM DB2), QSQLITE(SQLite) 및 QIBASE(Interbase)가 있다. 이 구동프로그람들중에서 일부는 Qt Open Source판에 포함되여있지 않다.

창조되는 련결은 응용프로그람의 기정자료기지련결로 되며 다른 자료기지가 지정되지 않았으면 Qt SQL클라스들에 의하여 사용된다.

둘째로, setDatabaseName(), setUserName(), setPassword() 그리고 setHostName()을 호출하여 련결정보를 초기화한다. QOCI8 (Oracle 8과 9) 구동프로그람에서는 TNS봉사이름을 setDatbaseName()에 넘겨야 한다. ODBC자료원천에 련결할 때 자료원천이름(DSN)을 setDatabaseName()호출에 사용하여야 한다.

셋째로, open()함수를 호출하여 자료기지를 열고 자료에 대한 호출을 준다. 이 호출이 실패하면 FALSE를 돌려주며 오유정보는 QSqlDatabase::lastError()로부터 얻을수 있다.

2) 여러 자료기지와 련결

여러 자료기지와의 련결은 둘째 인수가 련결을 구분하는 유일한 식별자인 QSqlDatabase::addDatabase()함수의 2인수형식을 사용하여 달성할수 있다.

다음의 실례에서 자체의 함수 createConnections()에로 련결들을 옮기고 몇가지 기초적인 오유조종을 추가하였다.

```
#define DB_SALES_DRIVER    "QPSQL7"
#define DB_SALES_DBNAME    "sales"
#define DB_SALES_USER      "salesperson"
#define DB_SALES_PASSWD    "salesperson"
#define DB_SALES_HOST      "database.domain.no"

#define DB_ORDERS_DRIVER   "QOCI8"
#define DB_ORDERS_DBNAME   "orders"
#define DB_ORDERS_USER     "orderperson"
#define DB_ORDERS_PASSWD   "orderperson"
#define DB_ORDERS_HOST     "database.domain.no"

bool createConnections();
```

connection.h에서 몇가지 상수들을 설정하고 createConnections()함수를 선언한다.

```
// sql/overview/connection.cpp로부터
  #include <qsqldatabase.h>
  #include "connection.h"

  bool createConnections()
  {

    QSqlDatabase *defaultDB = QSqlDatabase:: addDatabase( DB_SALES_DRIVER );
    defaultDB->setDatabaseName( DB_SALES_DBNAME );
    defaultDB->setUserName( DB_SALES_USER );
    defaultDB->setPassword( DB_SALES_PASSWD );
    defaultDB->setHostName( DB_SALES_HOST );
    if ( ! defaultDB->open() ) {
      qWarning( "Failed to open sales database: " + defaultDB->lastError().text() );
```

```
        return FALSE;
      }

      QSqlDatabase *oracle = QSqlDatabase::addDatabase( DB_ORDERS_DRIVER,
                                    "ORACLE" );
      oracle->setDatabaseName( DB_ORDERS_DBNAME );
      oracle->setUserName( DB_ORDERS_USER );
      oracle->setPassword( DB_ORDERS_PASSWD );
      oracle->setHostName( DB_ORDERS_HOST );
      if ( ! oracle->open() ) {
        qWarning( "Failed to open orders database: " + oracle->lastError().text() );
        return FALSE;
      }

      QSqlQuery q(QString::null, defaultDB);
      q.exec("create table people (id integer primary key, name char(40))");
      q.exec("create table staff (id integer primary key, forename char(40), "
             "surname char(40), salary float, statusid integer)");
      q.exec("create table status (id integer primary key, name char(30))");
      q.exec("create table creditors (id integer primary key, forename
char(40), " "surname char(40), city char(30))");
      q.exec("create table prices (id integer primary key, name char(40),
price float)");
      q.exec("create table invoiceitem (id integer primary key, "
             "pricesid integer, quantity integer, paiddate date)");

      QSqlQuery q2(QString::null, oracle);
      q2.exec("create table people (id integer primary key, name
char(40))");

      return TRUE;
    }
```

connection.cpp의 createConnections()함수에서 자료기지련결을 분리시키도록 선택하였다.

```
// sql/overview/create_connections/main.cpp로부터
    #include <qapplication.h>
    #include <qsqldatabase.h>
    #include "../connection.h"

    int main( int argc, char *argv[] )
    {
      QApplication app( argc, argv, FALSE );

      if ( createConnections() ) {
        // Databases successfully opened; get pointers to them:
        QSqlDatabase *oracledb = QSqlDatabase::database( "ORACLE" );
```

```
    // Now we can now issue SQL commands to the oracle
       connection
    // or to the default connection
  }

  return 0;
}
```

정적함수 QSqlDatabase::database()는 자료기지련결의 지적자를 제공하는곳이라면 어디서나 호출할수 있다. 파라메터없이 그것을 호출하면 기정련결을 돌려준다. 련결에 사용된 식별자(가령 우의 실례에서 "ORACLE")를 가지고 호출되면 지정된 련결의 지적자를 돌려준다.

Qt Designer에 의해 main.cpp를 창조한다면 그것은 실례와 같은 createConnections()함수를 포함하지 않는다. 이것은 Qt Designer안에서 보여주는 응용프로그람들에 자체의 자료기지련결함수를 실현하지 않으면 실행되지 않는다는것을 의미한다.

우의 코드에서 ODBC련결이 명명되지 않았으므로 기정련결로서 사용된다. QSqlDatabase는 addDatabase()정적함수에 의하여 되돌아온 지적자들의 소유권을관리한다. 관리하는 련결목록에서 자료기지를 삭제하기 위하여서는 QSqlDatabase::close()에 의하여 자료기지를 닫고 정적함수 QSqlDatabase::removeDatabase()에 의해 그것을 삭제한다.

4. QSqlQuery을 리용한 SQL지령의 실행

QSqlQuery클라스는 SQL지령들을 실행하기 위한 대면부를 제공한다. 또한 SELECT질문들의 결과모임을 항행하기 위한 함수들과 개별적인 레코드들과 마당값들을 검색하기 위한 함수들을 가진다.

다음 소절에서 서술하는 QSqlCursor클라스는 QSqlQuery로부터 계승되고 SQL지령들을 구성하는 고수준 대면부를 제공한다. QSqlCursor는 특히 화면우의 창문부품들과 통합하기 쉽다. SQL을 모르는 프로그람작성자들은 이 소절을 안전하게 뛰여넘어 QSqlCursor클라스를 사용할수 있다.

1) 일괄처리

기초하고있는 자료기지엔진이 일괄처리(transaction)를 지원한다면 QSqlDriver::hasFeature(QSqlDriver::Transactions)는 TRUE를 돌려준다. QSqlDatabase::transaction()에 의해 일괄처리를 초기화할수 있다. 이때 일괄처리문맥안에 실행하려는 SQL지령들을 따라보낸다. 그다음 QSqlDatabase::commit()나 QSqlDatabase::rollback()를 실행한다.

2) 기본열람

```
// sql/overview/basicbrowsing/main.cpp로부터
#include <qapplication.h>
  #include <qsqldatabase.h>
  #include <qsqlquery.h>
  #include "../connection.h"

  int main( int argc, char *argv[] )
  {
    QApplication app( argc, argv, FALSE );
```

```
if ( createConnections() ) {
  QSqlDatabase *oracledb = QSqlDatabase::database( "ORACLE" );
  // oracle자료기지에서 ODBC(기정)자료기지로 자료를 복사한다.
  QSqlQuery target;
  QSqlQuery query( "SELECT id, name FROM people", oracledb );
  if ( query.isActive() ) {
    while ( query.next() ) {
 target.exec( "INSERT INTO people ( id, name ) VALUES ( " +
  query.value(0).toString() +", '" + query.value(1).toString() +  "' )" );
    }
  }
}

return 0;
}
```

우의 실례에서는 머리부파일 qsqlquery.h를 추가하였다. 창조한 첫 질문 target는 기정자료기지를 사용하며 처음에는 비여있다. 둘째 질문 q에서 레코드를 검색하려는 "ORACLE"자료기지를 지정한다. 두 자료기지련결은 모두 앞에서 제시한 createConnections()함수안에서 설정되였다.

처음의 SELECT문을 창조한 다음에 질문이 성공적으로 실행되였는가를 확인하기 위하여 isActive()가 검사된다. next()함수는 질문결과들을 순환하는데 사용된다. value()함수는 마당들의 내용을 QVariant로서 돌려준다. 삽입은 target QSqlQuery객체를 사용하여 기정자료기지에 대한 질문들을 창조하고 실행함으로써 달성할수 있다.

이 실례와 이 절의 다른 모든 실례들은 소절 8에서 정의된 표들을 사용한다.

```
// sql/overview/basicbrowsing2/main.cpp로부터
    int count = 0;
    if ( query.isActive() ) {
      while ( query.next() ) {
 target.exec( "INSERT INTO people ( id, name ) VALUES ( " +
     query.value(0).toString() + ", '" + query.value(1).toString() +  "' )" );
 if ( target.isActive() )
   count += target.numRowsAffected();
      }
    }
```

우의 코드는 성과적으로 삽입된 레코드량을 소개한다. isActive()는 질문(실례로 삽입)이 실패하면 FALSE를 돌려준다. numRowsAffected()는 행수를 결정할수 없으면 실례로 질문이 실패하면 -1을 돌려준다.

3) 기본자료조작

```
// sql/overview/basicdatamanip/main.cpp로부터
 #include <qapplication.h>
 #include <qsqldatabase.h>
 #include <qsqlquery.h>
 #include "../connection.h"

 bool createConnections();
```

```
int main( int argc, char *argv[] )
{
  QApplication app( argc, argv, FALSE );

  int rows = 0;

  if ( createConnections() ) {
    QSqlQuery query( "INSERT INTO staff ( id, forename, surname, salary ) "
  "VALUES ( 1155, 'Ginger', 'Davis', 50000 )" );
    if ( query.isActive() ) rows += query.numRowsAffected() ;

    query.exec( "UPDATE staff SET salary=60000 WHERE id=1155" );
    if ( query.isActive() ) rows += query.numRowsAffected() ;

    query.exec( "DELETE FROM staff WHERE id=1155" );
    if ( query.isActive() ) rows += query.numRowsAffected() ;
  }

  return ( rows == 3 ) ? 0 : 1;
}
```

이 실례는 간단한 SQL자료조작언어(DML,data manipulation language)지령들을 설명한다. QSqlQuery구성자에서 자료기지를 지정하지 않았으므로 기정자료기지가 사용된다. 또한 QSqlQuery객체들은 CREATE TABLE, CREATE INDEX와 같은 SQL자료정의언어(DDL, data definition language)지령들을 실행하는데 사용된다.

4) 결과모임의 항행

일단 SELECT질문이 성공적으로 실행되면 질문기준에 대조되는 레코드들의 결과모임에 대한 접근이 가능하다. 이미 항행함수들중의 하나인 next()를 사용하였다. 이 함수는 레코드들에 차례로 접근하는데 독자적으로 사용될수 있다. 또한 QSqlQuery는 first()와 last(), prev()를 제공한다. 이러한 지령들다음에 isValid()를 호출하여 유효레코드에 있는가 검사할수 있다.

또한 seek()에 의하여 임의의 레코드에로 항행할수 있다. 자료모임의 첫 레코드는 령이다. 마지막 레코드의 번호는 size() - 1이다. 모든 자료기지가 SELECT질문의 크기를 제공하지 않으므로 이와 같은 경우에 size()는 -1을 돌려준다.

```
// sql/overview/navigating/main.cpp로부터
    if ( createConnections() ) {
      QSqlQuery query( "SELECT id, name FROM people ORDER BY name" );
      if ( ! query.isActive() ) return 1; // 질문실패
      int i;
      i = query.size(); // 이 실례에서 9개 레코드를 가진다. i == 9.
      query.first();  // 첫 레코드로 이동.
      i = query.at(); // i == 0
      query.last();   // 마지막 레코드로 이동.
      i = query.at(); // i == 8
      query.seek( query.size() / 2 ); // 중간레코드로 이동.
      i = query.at(); // i == 4
```

```
}
```

우의 실례는 사용중에 있는 항행함수들을 보여준다.

모든 구동프로그람들이 size()를 유지하지 않지만 그에 대하여 구동프로그람에 질문할수 있다.

```
QSqlDatabase* defaultDB = QSqlDatabase::database();
if ( defaultDB->driver()->hasFeature( QSqlDriver::QuerySize ) ) {
  // QSqlQuery::size()유지
}
else {
  // QSqlQuery::size()유지안됨
}
```

관심을 가지는 레코드를 탐색하였다면 그로부터 자료를 얻으려고 할수 있다.

```
// sql/overview/retrieve1/main.cpp로부터
  if ( createConnections() ) {
     QSqlQuery query( "SELECT id, surname FROM staff" );
     if ( query.isActive() ) {
       while ( query.next() ) {
         qDebug( query.value(0).toString() + ":
                 " + query.value(1).toString() );
       }
     }
  }
```

레코드모임을 차례로 순환하려고 한다면 요구되는 항행함수는 오직 next()뿐이다.

암시: lastQuery()함수는 실행된 마지막 질문의 본문을 돌려준다. 이것은 자기가 생각하는 질문이 실행되고있는가, 실제로 어느 질문이 실행되고있는가를 검사하는데 쓸모있다.

5. QSqlCursor의 리용

QSqlCursor클라스는 자체의 SQL을 작성하지 않고 SQL자료기지표나 보기에서 레코드를 열람하고 편집하기 위한 고수준대면부를 제공한다.

QSqlCursor는 QSqlQuery가 할수 있는 거의 모든것을 다할수 있지만 두가지 례외가 있다. 기정적으로 유표가 자료기지안에서 표나 보기를 표시하므로 QSqlCursor객체들은 새 레코드로 이동할 때마다 표나 보기안의 매 레코드의 모든 마당을 검색한다. 오직 몇개의 마당들만이 관련된다면 간단히 처리를 그것들로 제한하고 다른것들은 무시한다. 혹은 QSqlRecord::setGenerated()에 의한 마당들의 생성을 수동적으로 불가능하게 한다. 또 하나의 수법은 자기가 관심을 가지는 마당들만 표시하고 VIEW를 창조하는것인데 일부 자료기지는 편집할수 있는 보기를 지원하지 않는다. 그러므로 실제로 유표안의 모든 마당들을 검색하고싶지 않으면 대신에 QSqlQuery를 사용하여 자기 요구에 맞게 질문을 정의하여야 한다. 표 혹은 보기가 매 레코드를 유일하게 식별하는 1차색인을 가지도록 제공하는 QSqlCursor를 사용하여 레코드를 편집할수 있다. 이러한 조건이 만족되지 않으면 편집에 QSqlQuery를 사용할 필요가 있다.

QSqlCursor는 한번에 하나의 레코드에 대하여 조작한다. QSqlCursor에 의하여 삽입, 갱신, 삭제를 실행할 때마다 자료기지안에 있는 오직 하나의 레코드만이 영향을 받는다. 유표안의 레코드로 항행할 때 한번에 오직 하나의 레코드만 응용프로그람코드에서 사용할수 있다. 추가적으로 QSqlCursor는 자료기지안의 하나의 레코드에 변경이 이루어지게 하는데 쓰이는 하나의 개별적인 《편집완충기》를 관리한다. 편집완충기는

개별적인 기억공간안에서 유지되며 유표는 한개 레코드씩 움직일 때마다 변화되는 《항행완충기》의 영향을 받지 않는다.

QSqlCursor객체들을 사용하기전에 우선 자료기지련결을 창조하고 그것을 열어야 한다. 련결은 앞의 소절 3에서 서술하였다. 다음의 실례들에서는 이미 제시한 QSqlDatabase실례에서 정의된 createConnections()함수를 사용하여 련결이 창조되였다고 가정한다.

소절6절에서 자료인식창문부품에서 창문부품을 자료기지유표에 련결하는 방법을 보여준다. 일단 유표와 자료인식창문부품에 대한 지식을 둘다 가지면 QSqlCursor의 파생클라스만들기에 대하여 론의할수 있다.

QSqlCursor클라스는 qsqlcursor.h머리부파일을 요구한다.

1) 레코드검색

```
// sql/overview/retrieve2/main.cpp로부터
  #include <qapplication.h>
  #include <qsqldatabase.h>
  #include <qsqlcursor.h>
  #include "../connection.h"

  int main( int argc, char *argv[] )
  {
    QApplication app( argc, argv );

    if ( createConnections() ) {
      QSqlCursor cur( "staff" ); // Specify the table/view name
      cur.select(); // We'll retrieve every record
      while ( cur.next() ) {
        qDebug( cur.value( "id" ).toString() + ": " +
                cur.value( "surname" ).toString() + " " +
                cur.value( "salary" ).toString() );
      }
    }

    return 0;
  }
```

사용하려는 표나 보기를 지정하여 QSqlCursor객체를 창조한다. 기정이 아닌 자료기지를 사용해야 한다면 QSqlCursor구성자에서 그것을 지정할수 있다.

cur.select()호출에 의해 실행된 SQL은

```
SELECT staff.id, staff.forename, staff.surname, staff.salary, staff.statusid FROM staff
```

다음으로 cur.next()를 사용하여 이 select문으로부터 돌아온 레코드들을 순환한다. 마당값들은 QSqlQuery와 비슷한 방법으로 검색된다. 그러나 value()와 setValue()에 수값첨수가 아니라 마당이름을 넘기는것은 례외이다.

- 레코드의 정렬과 려과

검색하려는 레코드들의 부분모임을 지정하기 위하여 select()함수에 려파기준을 넘길수 있다. 돌아오는 매개 레코드는 려파기준을 만족시킨다. (려파기는 SQL문의 WHERE절에 대응된다.)

```
cur.select( "id > 100" );
```

이 select()호출은 다음의 SQL을 실행한다.

```
SELECT staff.id, staff.forename, staff.surname, staff.salary, staff.statusid
FROM staff WHERE staff.id > 100
```

이것은 id가 100이상인 직원만 검색한다.

선택된 레코드들을 검색하는것과 함께 때때로 돌려준 레코드들을 배렬순위로 지정하여야 한다. 이것은 정렬하려고 하는 마당(들)의 이름을 포함하는 QSqlIndex객체를 창조하고 select()호출에 이 객체를 넘기여 달성된다.

```
QSqlCursor cur( "staff" );
QSqlIndex nameIndex = cur.index( "surname" );
cur.select( nameIndex );
```

여기서는 하나의 마당 "surname"을 가지는 QSqlIndex객체를 창조한다. select()함수를 호출할 때 색인객체를 넘기는데 색인객체는 레코드들이 staff.surname에 의하여 정렬되여 돌아와야 한다는것을 지정한다. 색인객체의 매개 마당은 select문의 ORDER BY절에서 사용된다. 여기서 실행하는 SQL은 다음과 같다.

```
SELECT staff.id, staff.forename, staff.surname, staff.salary, staff.statusid
FROM staff ORDER BY staff.surname ASC
```

레코드들의 부분모임검색을 결합하고 결과를 정렬하기는 쉽다.

```
cur.select( "staff.surname LIKE 'A%'", nameIndex );
```

려과문자렬(WHERE절)과 함께 정렬하려는 QSqlIndex객체(ORDER BY절)을 넘긴다. 이것은 다음과 같은 문으로 수행한다.

```
SELECT staff.id, staff.forename, staff.surname, staff.salary, staff.statusid
FROM staff WHERE staff.surname LIKE 'A%' ORDER BY staff.surname ASC
```

하나이상의 마당을 정렬하려면 여러 마당을 포함하는 색인을 창조할수 있다. 커지는 자모순과 작아지는 자모순은QSqlIndex::setDescending()에 의하여 설정할수 있으며 기정값은 커지는 자모순이다.

```
// sql/overview/order1/main.cpp로부터
QSqlCursor cur( "staff" );
    QStringList fields = QStringList() << "surname" << "forename";
    QSqlIndex order = cur.index( fields );
    cur.select( order );
    while ( cur.next() ) {
```

여기서 정렬하려는 마당들을 포함하는 문자렬목록을 창조하는데 그것들을 차례로 사용한다. 그다음 이 마당들에 기초하는 QSqlIndex객체를 창조하고 끝으로 이 색인을 리용하여 select()호출을 실행한다. 이것은 다음 문으로 실행한다.

```
SELECT staff.id, staff.forename, staff.surname, staff.salary, staff.statusid
FROM staff ORDER BY staff.surname ASC, staff.forename ASC
```

지정된 기준과 일치하는 마당을 가지는 레코드들을 검색해야 한다면 어떤 색인에 기초한 려파기를 창조할수 있다.

```
// sql/overview/order2/main.cpp로부터
QSqlCursor cur( "staff" );
    QStringList fields = QStringList() << "id" << "forename";
```

```
QSqlIndex order = cur.index( fields );
QSqlIndex filter = cur.index( "surname" );
cur.setValue( "surname", "Bloggs" );
cur.select( filter, order );
while ( cur.next() ) {
```

이것은 다음 문을 실행한다.

SELECT staff.id, staff.forename, staff.surname, staff.salary, staff.statusid
FROM staff WHERE staff.surname='Bloggs' ORDER BY staff.id ASC, staff.forename ASC

QSqlIndex객체 order는 두개의 마당 id와 forename을 포함하는데 그것들은 결과들을 순서화하는데 쓰인다. QSqlIndex객체 filter는 하나의 마당 surname을 포함한다. 색인이 select()함수에 려파기로서 넘어갈 때 려파기안의 매 마당에 대하여*fieldname=value*부분절이 창조되는데 그 마당의 현재유표값으로부터 값이 취해진다. setValue()에 의하여 사용된 값이 요구되는 값이라는것을 보증한다.

- 자료발취

```
// sql/overview/extract/main.cpp로부터
    QSqlCursor cur( "creditors" );
    QStringList orderFields = QStringList() << "surname" << "forename";
    QSqlIndex order = cur.index( orderFields );

    QStringList filterFields = QStringList() << "surname" << "city";
    QSqlIndex filter = cur.index( filterFields );
    cur.setValue( "surname", "Chirac" );
    cur.setValue( "city", "Paris" );

    cur.select( filter, order );

    while ( cur.next() ) {
      int id = cur.value( "id" ).toInt();
      QString name = cur.value( "forename" ).toString() + " " +
      cur.value( "surname" ).toString();
      qDebug( QString::number( id ) + ": " + name );
    }
```

이 실례에서는 creditors표에 유표를 창조하는것으로시작한다. 두개의 QSqlIndex객체를 창조한다. 첫째 객체 order는 orderFields문자렬목록으로부터 창조된다. 둘째 객체 filter는 filterFields문자렬목록으로부터 창조된다. 려파기에서 사용되는 두개의 마당 surname과 city의 값들을 우리가 관심을 가지는 값들로 설정한다. 이제는 select()를 호출하여다음과 같은 SQL을 생성하고 실행한다.

SELECT creditors.city, creditors.surname, creditors.forename, creditors.id
FROM creditors
WHERE creditors.surname = 'Chirac' AND creditors.city = 'Paris'
ORDER BY creditors.surname ASC, creditors.forename ASC

려파기마당들은 WHERE부분에서 쓰인다. 마당들의 값들은 그 마당들에 대한 유표의 현재값들로부터 얻어지고 setValue()를 호출하여 자체로 이 값들을 설정한다.

order마당들은 ORDER BY부분에서 쓰인다.

그다음 일치하는 매개 레코드(있는 경우)를 순환한다. id와 forename, surname 마당들의 내용을 검색하고 그것들을 처리함수에 넘긴다. 이 실례에서는단순히 qDebug()호출에 넘긴다.

2) 레코드조작

표나 보기가 매 레코드를 유일하게 식별하는 1차색인을 가지도록 제공하는 QSqlCursor를 사용하여 레코드를 표나 보기에서 삽입, 갱신, 삭제할수 있다. 그렇지 않으면 QSqlQuery를 대신에 사용하여야 한다. (모든 자료기지가 편집가능한 보기를 지원하는것은 아니다.)

매 유표는 모든 편집조작들(삽입, 갱신, 삭제)에 의하여 사용되는 내부《편집완충기》를 가진다. 편집과정은 매 조작에서 같은데 관련한 완충기의 지적자를 얻고 setValue()를 호출하여 요구하는 값으로 완충기를 초기화하며 insert()나 update()나 del()을 호출하여 필요한 조작을 수행한다. 실례로 유표를 사용하여 레코드를 삽입할 때 primeInsert()를 호출하여 편집완충기의 지적자를 얻은 다음 이 완충기에서 setValue()를 호출하여 매개 마당값을 설정한다. 그다음 QSQlCursor::insert()를 호출하여 자료기지에 편집완충기의 내용을 삽입한다. 마찬가지로 레코드를 갱신(혹은 삭제)할 때 편집완충기안의 마당의 값들은 자료기지의 레코드를 갱신(혹은 삭제)하는데 사용된다. 편집완충기는 유표항행함수들에 의하여 어떤 영향도 받지 않는다 . setValue()에 문자렬값을 넘긴다면 단일인용표는 SQL의 특수문자이므로 그것들은 확장되여 단일인용표쌍으로 바뀐다.

primeInsert()와 primeUpdate(), primeDelete()메쏘드들은 모두 내부편집완충기의 지적자를 돌려준다. 매개 메쏘드는 되돌아가기전에 편집완충기에 대한 서로 다른 조작들을 잠재적으로 수행할수 있다. 기정적으로 QSqlCursor::primeInsert()는 편집완충기안의 모든 마당값들을 지운다(QSqlRecord::clearValues()참고). QSqlCursor::primeUpdate()와 QSqlCursor::primeDelete()는 둘다 되돌아가기전에 유표의 현재내용으로 편집완충기를 초기화한다. 3개의 함수들이 모두 가상함수이므로 그 동작을 재정의할수 있다. 실례로 primeInsert()를 재정의하여 편집완충기안의 마당들에 자동적으로 번호를 붙인다. 자료인식 사용자대면부조종요소들은 련결할수 있는 신호들(실례로 primeInsert())를 발생하며 이것들은 적당한 완충기의 지적자를 넘기므로 파생클라스화가 필요없을수도 있다. (파생클라스화에 대한 자세한 정보는 소절7을 보고 primeInsert()신호와의 련결에 대해서는 《Qt프로그람개발도구》 1장를 보시오.)

유표에 대하여 insert(), update() 혹은 del()을 호출할 때 그것은 무효화되고 유효레코드에 더는 위치하지 않는다. insert(), update() 혹은 del()을 수행한 후에 다른 레코드로 이동해야 한다면 순수한select()호출을 만들어야 한다. 이것은 자료기지에 대한 변경이 유표에 정확하게 반영되도록 한다.

① 레코드삽입

```
// sql/overview/insert/main.cpp로부터
    QSqlCursor cur( "prices" );
     QStringList names =
          QStringList()<<"Screwdriver"<<"Hammer"<<"Wrench"<<"Saw";
     int id = 20;
     for ( QStringList::Iterator name = names.begin();
             name != names.end(); ++name ) {
       QSqlRecord *buffer = cur.primeInsert();
       buffer->setValue( "id", id );
```

```
            buffer->setValue( "name", *name );
            buffer->setValue( "price", 100.0 + (double)id );
            count += cur.insert();
            id++;
        }
```

이 실례에서는 prices표에서 유표를 창조한다. 다음으로 순환하려는 제품이름목록을 창조한다. 매번 순환할 때마다 유표의 primeInsert()메쏘드를 호출한다. 이 메쏘드는 모든 마당들이 NULL로 설정되여있는 QSqlRecord완충기의 지적자를 돌려준다. (QSqlCursor::primeInsert()는 가상이므로 파생클라스에 의해 전용화될수 있다.) 다음에 값을 요구하는 매 마당에 대하여 setValue()를 호출한다. 끝으로 insert()를 호출하여 레코드를 삽입한다. insert()호출은 삽입한 렬수를 돌려준다.

primeInsert()호출로부터 QSqlRecord객체의 지적자를 얻었다. QSqlRecord객체들은 단일레코드에 대한 자료와 레코드에 대한 메타자료를 보유할수 있다. 실천에서 QSqlRecord와의 대부분의 교제는 이 실례와 다음 실례에서 보여주는것처럼 단순한 value()와 setValue()호출들로 이루어진다.

② 레코드의 갱신

```
// sql/overview/update/main.cpp로부터
      QSqlCursor cur( "prices" );
       cur.select( "id=202" );
       if ( cur.next() ) {
         QSqlRecord *buffer = cur.primeUpdate();
         double price = buffer->value( "price" ).toDouble();
         double newprice = price * 1.05;
         buffer->setValue( "price", newprice );
         cur.update();
       }
```

이 실례는 prices표에 유표를 창조하는것으로 시작한다. select()호출에 의해 갱신하려는 레코드를 선택하고 next()호출에 의해 거기로 이동한다. primeUpdate()를 호출하여 현재 레코드의 내용으로 채워지는 완충기에로의 QSqlRecord지적자를 얻는다. price마당값을 검색하고 새로운 단가를 계산하고 price마당을 새로 계산한 값으로 설정한다. 끝으로 update()를 호출하여 레코드를 갱신한다. update()호출은 갱신된 행수를 돌려준다.

꼭같은 갱신을 여러개 실현하여야 한다면 실례로 단가목록의 매개 항목의 단가를 증가시키려면 QSqlQuery을 가지는 단일한 SQL문을 사용하는것이 더 효과있다. 실례로

```
    QSqlQuery query( "UPDATE prices SET price = price * 1.05"  );
```

③ 레코드의 삭제

```
// sql/overview/delete/main.cpp로부터
 QSqlCursor cur( "prices" );
       cur.select( "id=999" );
       if ( cur.next() ) {
         cur.primeDelete();
         cur.del();
```

레코드를 삭제하기 위해서는 삭제하려는 레코드를 선택하고 거기로 항행한다. 그다음 primeDelete()를 호출하여 선택된 레코드의 주열쇠(이 실례에서 prices.id마당)로 유표를 설정하고 QSqlCursor::del()을 호출하여 그것을 삭제한다.

update()에서처럼 어떤 일반적인 기준에 따르는 다중삭제가 필요하다면 하나의

SQL문을 사용하는것이 더 효과있다. 실례로

```
QSqlQuery query( "DELETE FROM prices WHERE id >= 2450 AND id <= 2500" );
```

6. 자료인식창문부품

자료인식창문부품(data-aware widget)은 Qt사용자대면부에 자료기지를 련결하는 간단하고 강력한 수단을 제공한다. 자료인식창문부품을 창조하고 조작하는 가장 쉬운 방법은 Qt Designer를 사용하는것이다. 다음의 실례와 설명은 순수 프로그람적수법을 좋아하는 사람들에게 하나의 소개로 된다. (《Qt프로그람개발도구》의 1장 7절과 거기에 첨부한 실례들에서 추가정보를 제공한다.)

1) 자료인식표들

```
// sql/overview/table1/main.cpp로부터
  #include <qapplication.h>
  #include <qsqldatabase.h>
  #include <qsqlcursor.h>
  #include <qdatatable.h>
  #include "../connection.h"

  int main( int argc, char *argv[] )
  {
    QApplication app( argc, argv );

    if ( createConnections() ) {
      QSqlCursor staffCursor( "staff" );
      QDataTable *staffTable = new QDataTable( &staffCursor, TRUE );
      app.setMainWidget( staffTable );
      staffTable->refresh();
      staffTable->show();

      return app.exec();
    }

    return 0;
  }
```

자료인식표는 qdatatable.h와 qsqlcursor.h머리부파일을 요구한다. 응용프로그람객체를 창조하고 createConnections()을 호출하여 유표를 창조한다. 유표의 지적자를 넘기여 QDataTable을 창조하고 autoPopulate기발을 TRUE로 설정한다. 다음으로 QDataTable를 기본창문부품으로 만들고 refresh()를 호출하여 거기에 자료를 채우고 show()를 호출하여 표시한다.

autoPopulate기발은 유표에 기초하여 렬들을 창조하여야 하는가를 QDataTable에 알려준다. autoPopulate는 표에 자료를 적재하는데 영향을 주지 않으며 그것은 refresh()함수에 의하여 달성된다.

```
// sql/overview/table2/main.cpp로부터
      QSqlCursor staffCursor( "staff" );
      QDataTable *staffTable = new QDataTable( &staffCursor );
```

```
app.setMainWidget( staffTable );

staffTable->addColumn( "forename", "Forename" );
staffTable->addColumn( "surname",  "Surname" );
staffTable->addColumn( "salary",   "Annual Salary" );

QStringList order = QStringList() << "surname" << "forename";
staffTable->setSort( order );

staffTable->refresh();
staffTable->show();
```

기본창문부품으로 만들려는 빈 QDataTable을 창조한 다음 현시하려는 순서로 필요한 렬들을 수동적으로 추가한다. 매 렬에 대하여 마당이름과 함께 선택적으로 현시표식을 지정한다.

또한 표안의 렬들을 정렬하려고 하며 이것은 자체로 유표에 정렬을 적용함으로써 달성된다.

일단 모든것이 설정되면 refresh()를 호출하여 자료기지로부터 자료를 적재하고 show()를 호출하여 창문부품을 표시한다.

QDataTables은 시각적인 렬들만 검색하며 이것은 아주 큰 표들을 최소기억원가로서 매우 고속으로 현시하게 한다. (구동프로그람에 의존한다.)

2) 자료인식폼의 창조

자료인식폼의 창조는 매 마당을 개별적으로 고찰해야 하므로 자료인식표를 사용하는것보다 더 복잡하다. 아래의 대부분의 코드는 Qt Designer에 의하여 자동생성된다.

- 레코드의 현시

// sql/overview/form1/main.cpp로부터

```
#include <qapplication.h>
#include <qdialog.h>
#include <qlabel.h>
#include <qlayout.h>
#include <qlineedit.h>
#include <qsqldatabase.h>
#include <qsqlcursor.h>
#include <qsqlform.h>
#include "../connection.h"

class FormDialog : public QDialog
{
  public:
    FormDialog();
};

FormDialog::FormDialog()
{
  QLabel *forenameLabel   = new QLabel( "Forename:", this );
  QLabel *forenameDisplay = new QLabel( this );
  QLabel *surnameLabel  = new QLabel( "Surname:", this );
```

```
  QLabel *surnameDisplay  = new QLabel( this );
  QLabel *salaryLabel   = new QLabel( "Salary:", this );
  QLineEdit *salaryEdit   = new QLineEdit( this );

  QGridLayout *grid = new QGridLayout( this );
  grid->addWidget( forenameLabel,   0, 0 );
  grid->addWidget( forenameDisplay,   0, 1 );
  grid->addWidget( surnameLabel,    1, 0 );
  grid->addWidget( surnameDisplay,  1, 1 );
  grid->addWidget( salaryLabel,     2, 0 );
  grid->addWidget( salaryEdit,    2, 1 );
  grid->activate();

  QSqlCursor staffCursor( "staff" );
  staffCursor.select();
  staffCursor.next();

  QSqlForm sqlForm( this );
  sqlForm.setRecord( staffCursor.primeUpdate() );
  sqlForm.insert( forenameDisplay, "forename" );
  sqlForm.insert( surnameDisplay, "surname" );
  sqlForm.insert( salaryEdit, "salary" );
  sqlForm.readFields();
}

int main( int argc, char *argv[] )
{
  QApplication app( argc, argv );

  if ( ! createConnections() ) return 1;

  FormDialog *formDialog = new FormDialog();
  formDialog->show();
  app.setMainWidget( formDialog );

  return app.exec();
}
```

필요한 창문부품들을 위한 머리부파일들을 포함한다. 보통 qsqldatabase.h과 qsqlcursor.h를 포함하지만 이번에는 qsqlform.h를 추가한다.

자체의 FormDialog클라스를 가지고 QDialog클라스의 파생클라스를 만드므로 폼은 대화칸으로 제공된다. 사용자가 월생활비를 변경하는데 QLineEdit를 사용한다. 모든 창문부품들을 살창상태로 배치한다.

staff표에 유표를 창조하고 레코드들을 모두 선택하고 첫 레코드에로 이동한다.

이제는 QSqlForm객체를 창조하고 QSqlForm의 레코드완충기를 유표의 갱신완충기로 설정한다. 자료를 인식하게 하려는 매개 창문부품에 대하여 창문부품의 지적자 그리고 관련된 마당이름을 QSqlForm에 삽입한다. 끝으로 readFields()를 호출하여 유표

의 완충기를 거쳐서 자료기지의 자료로 창문부품들을 채워넣는다.

- 자료폼에서 레코드현시

QDataView는 읽기전용 QSqlForm을 가질수 있는 창문부품이다. QSqlForm과 함께 이것은 처리부 refresh(QSqlRecord *)를 제공하므로 레코드의 구체적인 보기를 현시할수 있는 QDataTable과 함께 간단히 련결될수 있다.

```
connect( myDataTable, SIGNAL( currentChanged( QSqlRecord* ) ),
     myDataView, SLOT( refresh( QSqlRecord* ) ) );
```

- 레코드편집

이 실례는 앞의것과 비슷하므로 차이에 초점을 둔다.

```
// sql/overview/form2/main.h로부터
  class FormDialog : public QDialog
  {
    Q_OBJECT
    public:
      FormDialog();
      ~FormDialog();
    public slots:
      void save();
    private:
      QSqlCursor staffCursor;
      QSqlForm *sqlForm;
      QSqlIndex idIndex;
  };
```

save처리부는 사용자가 갱신을 확인하기 위해 누르는 단추에 사용된다. 또한 QSqlCursor과 QSqlForm의 지적자들을 보유하므로 구성자밖에서 그것들을 호출할수 있다.

```
staffCursor.setTrimmed( "forename", TRUE );
staffCursor.setTrimmed( "surname",  TRUE );
```

본문마당들에 대하여 setTrimmed()를 호출하여 그 마당들을 채우는데 사용된 공간들은 그 마당들이 검색될 때 제거되도록 할수 있다.

배렬과 확인과 같이 마당들에 적용할수 있는 속성들은 관례적인 방법으로 (실례로 QLineEdit::setAlignment()와 QLineEdit::setValidator()를 호출함으로써) 달성할수 있다.

```
QLineEdit   *forenameEdit  = new QLineEdit( this );
QPushButton *saveButton  = new QPushButton( "&Save", this );
connect( saveButton, SIGNAL(clicked()), this, SLOT(save()) );
```

FormDialog구성자는 앞의 실례와 비슷하다. forename과 surname창문부품들을 QLineEdit들로 변경하여 편집가능하게 만들고 사용자가 마우스로 선택하여 갱신을 보관할수 있는 QPushButton을 추가한다.

```
grid->addWidget( saveButton,  3, 0 );
```

보관단추를 포함하고있는 살창에 여분의 렬을 하나 추가한다.

```
idIndex = staffCursor.index( "id" );
staffCursor.select( idIndex );
staffCursor.first();
```

QSqlIndex객체를 창조한 다음 색인을 사용하여 select()를 실행한다. 그다음 결과모임의 첫 레코드로 이동한다.

```
sqlForm = new QSqlForm( this );
sqlForm->setRecord( staffCursor.primeUpdate() );
```

새로운 QSqlForm객체를 창조하고 그 레코드완충기를 유표의 갱신완충기로 설정한다.

```
sqlForm->insert( forenameEdit, "forename" );
sqlForm->insert( surnameEdit, "surname" );
sqlForm->insert( salaryEdit, "salary" );
sqlForm->readFields();
```

이제는 완충기의 마당을 QLineEdit조종요소들에 련결한다. (앞의 실례에서는 유표의 마당들을 련결하였다.) 편집조종요소들은 앞에서처럼 readFields()호출에 의해 채워진다.

```
FormDialog::~FormDialog()
{
}
```

해체자에서는 창문부품들이나 QSqlForm에 대하여 걱정하지 말아야 한다. 왜냐하면 그것들은 폼의 자식들이고 정확한 순간에 Qt에 의하여 삭제되기때문이다.

```
void FormDialog::save()
{
  sqlForm->writeFields();
  staffCursor.update();
  staffCursor.select( idIndex );
  staffCursor.first();
}
```

끝으로 사용자가 보관단추를 누르는 순간을 위하여 보관기능을 추가한다. writeFields()호출에 의하여 창문부품들로부터 QSqlRecord완충기에로 자료를 써넣는다. 그다음 유표의 update()함수에 의해 갱신판의 레코드로 자료기지를 갱신한다. 이 시점에서 유표가 더는 유효레코드에 위치하지 않으므로 QSqlIndex를 리용하여 select()호출을 재발생하여 첫 레코드로 이동한다.

QDataBrowser와 QDataView는 우와 같은 기능을 대체로 제공하는 창문부품들이다. QDataBrowser는 유표의 레코드편집과 항행을 허용하는 자료폼을 제공한다. QDataView는 유표안의 자료나 자료기지레코드를 위한 읽기전용폼을 제공한다.

3) 사용자정의편집기창문부품들

QSqlForm은 QSqlPropertyMap를 사용하여 창문부품들과 자료기지마당들사이의 자료의 전송을 조종한다. 사용자정의창문부품들도 역시 자료전송에 사용될 사용자정의창문부품의 속성들에 대한 정보를 포함하는 속성략도를 설치함으로써 폼에서 사용될수 있다.

이 실례는 앞의 소절의 form2실례에 기초하고있으므로 여기서는 차이점만 설명한다. 완전한 원천은 sql/overview/custom1/main.h와 sql/overview/custom1/main.cpp에 있다.

```
class CustomEdit : public QLineEdit
{
  Q_OBJECT
  Q_PROPERTY( QString upperLine READ upperLine WRITE setUpperLine )
  public:
    CustomEdit( QWidget *parent=0, const char *name=0 );
    QString upperLine() const;
```

```
    void setUpperLine( const QString &line );
  public slots:
    void changed( const QString &line );
  private:
    QString upperLineText;
};
```

QLineEdit의 간단한 파생클라스를 창조하고 대문자판의 본문을 유지하는 upperLineText속성을 추가하였다. 또한 처리부 changed()를 창조한다.

```
QSqlPropertyMap *propMap;
```

FormDialog의 비공개자료에 속성략도의 지적자를 추가하기 위해 속성략도를 사용하고있다.

```
CustomEdit::CustomEdit( QWidget *parent, const char *name ) :
  QLineEdit( parent, name )
{
  connect( this, SIGNAL(textChanged(const QString &)),
      this, SLOT(changed(const QString &)) );
}
```

CustomEdit구성자에서는 QLineEdit구성자를 사용하며 textChanged신호와 자체의 changed처리부사이에 련결을 추가한다.

```
void CustomEdit::changed( const QString &line )
{
  setUpperLine( line );
}
```

changed()처리부는setUpperLine()함수를 호출한다.

```
void CustomEdit::setUpperLine( const QString &line )
{
  upperLineText = line.upper();
  setText( upperLineText );
}
```

setUpperLine()함수는 본문의 대문자사본을 upperLineText완충기에 배치한 다음 창문부품의 본문을 이 본문으로 설정한다.

CustomEdit클라스는 입력본문이 늘 대문자라는것을 담보하며 속성략도에 의하여 자료기지마당들에 CustomEdit실례들을 직접 련결하는데 쓰이는 속성을 제공한다.

```
CustomEdit *forenameEdit  = new CustomEdit( this );
CustomEdit *surnameEdit  = new CustomEdit( this );
```

앞에서와 같이 FormDialog을 사용하지만 이번에는 2개의 QLineEdit창문부품들을 자체의 CustomEdit창문부품들로 바꾼다.

살창형태의 배치관리자와 유표설정은 이전과 같다.

```
propMap = new QSqlPropertyMap;
propMap->insert( forenameEdit->className(), "upperLine" );
```

히프에 새로운 속성략도를 창조하고 CustomEdit클라스와 그 upperLine속성을 속성략도에 등록한다.

```
sqlForm = new QSqlForm( this );
sqlForm->setRecord( staffCursor->primeUpdate() );
sqlForm->installPropertyMap( propMap );
```

마지막 변경은 일단 QSqlForm이 창조되면 QSqlForm에 속성략도를 설치하는것이

다. 이것은 FormDialog가 소유하는 QSqlForm에 속성략도를 기억하게 하므로 Qt는 적당한 순간에 그것들을 삭제한다.

이 실례의 동작은 앞의 실례와 비슷하지만 forename과 surname마당들이 CustomEdit창문부품을 사용하므로 대문자로 된다.

- 표의 사용자정의편집기창문부품

표에서 사용자정의편집기창문부품을 허용하기 위해서는 QSqlEditorFactory를 재정의하여야 한다. 다음의 실례에서는 QComboBox와 QSqlEditorFactory파생클라스에 기초하는 사용자정의편집기를 창조하고 QDataTable이 사용자정의편집기를 사용하는 방법을 보여준다.

```
// sql/overview/table3/main.h로부터
  class StatusPicker : public QComboBox
  {
    Q_OBJECT
    Q_PROPERTY( int statusid READ statusId WRITE setStatusId )
    public:
      StatusPicker( QWidget *parent=0, const char *name=0 );
      int statusId() const;
      void setStatusId( int id );
    private:
      QMap< int, int > index2id;
  };
```

statusid속성을 창조하고 그것을 위한 READ와 WRITE메쏘드들을 정의한다. status표안의 statusid는 복합칸의 색인들과 다르므로 QMap를 창조하여 복합칸색인들을 그 복합칸에 렬거하는 statusid들로 넘기기한다.

```
  class CustomSqlEditorFactory : public QSqlEditorFactory
  {
    Q_OBJECT
    public:
      QWidget *createEditor( QWidget *parent, const QSqlField *field );
  };
```

또한 createEditor()는 재정의해야 할 유일한 함수이므로 이 함수를 선언하는 QSqlEditorFactory의 파생클라스를 만들어야 한다.

```
// sql/overview/table3/main.cpp로부터
  StatusPicker::StatusPicker( QWidget *parent, const char *name )
    : QComboBox( parent, name )
  {
    QSqlCursor cur( "status" );
    cur.select( cur.index( "name" ) );

    int i = 0;
    while ( cur.next() ) {
      insertItem( cur.value( "name" ).toString(), i );
      index2id[i] = cur.value( "id" ).toInt();
      i++;
    }
```

StatusPicker의 구성자에서는 name마당에 의하여 색인화되는 status표에 유표를

창조한다. 그다음 복합칸에 매개 이름을 삽입하면서 status표의 매개 레코드를 순환한다. 복합칸색인과 같은 QMap색인을 사용하여 index2id QMap에 매개 이름에 대한 statusid를 보관한다.

```
int StatusPicker::statusId() const
{
  return index2id[ currentItem() ];
}
```

statusid속성 READ함수는 복합칸색인들을 statusid들로 사영하는 index2id QMap안에서 현재 선택된 항목들에 대하여 복합칸색인의 검색기능을 포함한다.

```
void StatusPicker::setStatusId( int statusid )
{
  QMap<int,int>::Iterator it;
  for ( it = index2id.begin(); it != index2id.end(); ++it ) {
    if ( it.data() == statusid ) {
      setCurrentItem( it.key() );
      break;
    }
  }
}
```

statusId()함수는 statusid속성의 WRITE함수를 실현한다. QMap에 대하여 반복자를 창조하고 index2id QMap를 순환한다. 매개 index2id요소들의 자료(statusid)를 id파라메터들의 값과 비교한다. 일치하는것이 있으면 복합칸의 현재 항목을 index2id요소의 열쇠(복합칸색인)로 설정하고 순환고리를 벗어난다.

사용자가 QDataTable의 status마당을 편집할 때 status표로부터 취해진 유효상태 이름들이 복합칸에 표시된다. 그러나 표시된 상태는 아직 본래의 statusid이다. 마당을 편집하지 않을 때 상태이름을 현시하려면 QDataTable의 파생클라스를 만들고 paintField()함수를 재정의하여야 한다.

```
// sql/overview/table4/main.h로부터
  class CustomTable : public QDataTable
  {
    Q_OBJECT
  public:
    CustomTable(QSqlCursor *cursor, bool autoPopulate = FALSE,
        QWidget * parent = 0, const char * name = 0 ) :
      QDataTable( cursor, autoPopulate, parent, name ) {}
    void paintField(QPainter * p, const QSqlField* field, const QRect & cr, bool );

  };
```

아무것도 변경하지 않고 원래의 QDataTable구성자를 단순히 호출한다. 또한 paintField함수를 선언한다.

```
// sql/overview/table4/main.cpp로부터
  void CustomTable::paintField( QPainter * p, const QSqlField* field,
const QRect & cr, bool b)
  {
    if ( !field )
      return;
```

```
    if ( field->name() == "statusid" ) {
      QSqlQuery query( "SELECT name FROM status WHERE id=" +
    field->value().toString() );
      QString text;
      if ( query.next() ) {
        text = query.value( 0 ).toString();
      }
      p->drawText( 2,2, cr.width()-4, cr.height()-4, fieldAlignment( field ), text );
    }
    else {
      QDataTable::paintField( p, field, cr, b) ;
    }
```

paintField코드는 QDataTable의 원천코드에 기초하고있다. 3가지 변경이 필요하다. 첫째로, if절 field->name() == "statusid"를 추가하고 간단히 QSqlQuery에 의해 id의 본문값을 찾는다. 둘째로, 웃준위클라스를 호출하여 다른 마당들을 조종한다. 끝으로 staffTable을 QDataTable로부터 CustomTable로 변경하는 main함수가 있어야 한다.

7. QSqlCursor의 파생클라스작성

// sql/overview/subclass1/main.cpp로부터

```
  #include <qapplication.h>
  #include <qsqldatabase.h>
  #include <qsqlcursor.h>
  #include <qdatatable.h>
  #include "../connection.h"

  int main( int argc, char *argv[] )
  {
    QApplication app( argc, argv );

    if ( createConnections() ) {
      QSqlCursor invoiceItemCursor( "invoiceitem" );

      QDataTable            *invoiceItemTable            =            new
QDataTable( &invoiceItemCursor );

      app.setMainWidget( invoiceItemTable );

      invoiceItemTable->addColumn( "pricesid", "PriceID" );
      invoiceItemTable->addColumn( "quantity", "Quantity" );
      invoiceItemTable->addColumn( "paiddate", "Paid" );

      invoiceItemTable->refresh();
      invoiceItemTable->show();

      return app.exec();
```

```
        }

        return 1;
    }
```
이 실례는 처음에 제시한 table1실례와 거의 비슷하다. 유표를 창조하고 마당들과 그 형의 표식자들을 QDataTable에 추가하고 refresh()를 호출하여 자료를 적재하고 show()를 호출하여 창문부품을 표시한다.

그렇지만 이 실례는 충분하지 않다. 표이름설정과 이 표에 유표가 요구될 때마다 마당들의 사용자정의특성을 설정하는것은 시끄러운 일이다. 그리고 pricesid보다도 제품의 이름을 현시하면 더 좋을수 있다. 제품의 가격과 량을 알고있으므로 제품의 가격과 매개 invoiceitem의 가격을 표시할수 있다. 끝으로 사용자가 새 레코드를 추가할 때 일부 값들을 기정으로 한다면 쓸모있다. (혹은 주열쇠도 본질적이다.)

```
// sql/overview/subclass2/main.h로부터
    class InvoiceItemCursor : public QSqlCursor
    {
        public:
            InvoiceItemCursor();
    };
```
개별적인 머리부파일을 창조하고 QSqlCursor파생클라스를 만든다.

```
// sql/overview/subclass2/main.cpp로부터
    InvoiceItemCursor::InvoiceItemCursor() : QSqlCursor( "invoiceitem" )
    {
        // NOOP
    }
```
클라스의 구성자에서는 표의 이름을 가지고 QSqlCursor구성자를 호출한다. 이 단계에서 다른 특성을 추가하지 않는다.

```
        InvoiceItemCursor invoiceItemCursor;
```
invoiceitem표에 유표를 요구할 때마다 QSqlCursor대신에 InvoiceItemCursor를 창조할수 있다.

아직은 pricesid가 아니라 제품id를 표시해야 한다.

```
// sql/overview/subclass3/main.h로부터
        protected:
            QVariant calculateField( const QString & name );
```
머리부파일에서의 변경은 극히 적다. 단지 calculateField()함수를 재정의하므로 그 선언을 추가한다.

```
// sql/overview/subclass3/main.cpp로부터
    InvoiceItemCursor::InvoiceItemCursor()                              :
QSqlCursor( "invoiceitem" )
    {
        QSqlFieldInfo productName( "productname", QVariant::String );
        append( productName );
        setCalculated( productName.name(), TRUE );
    }

    QVariant InvoiceItemCursor::calculateField( const QString & name )
    {
```

```
    if ( name == "productname" ) {
      QSqlQuery query( "SELECT name FROM prices WHERE id=" +
    field( "pricesid" )->value().toString() );
      if ( query.next() )
        return query.value( 0 );
    }

    return QVariant( QString::null );
  }
```

InvoiceItemCursor구성자를 변경하였다. 이제 productname이라는 QSqlField를 새로 창조하고 이것을 InvoiceItemCursor의 마당모임에 추가한다. productname에 대하여 setCalculated()를 호출하여 그것을 계산된 마당으로서 식별한다. setCalculated()에로의 첫 인수는 마당이름이고 둘째 인수는 마당값을 얻기 위하여 calculateField()가 호출해야 한다는것을 TRUE로 표시하는 bool형이다.

```
        invoiceItemTable->addColumn( "productname", "Product" );
```

addColumn()에 의해 폼에 새 마당들을 추가하고 그 현시마당들을 설정한다.

자체의 calculateField()함수를 정의해야 한다. 실례자료기지에서 invoiceitem표의 pricesid는 prices표에로의 외부열쇠이다. pricesid를 사용하여 prices표에 대하여 질문을 실행함으로써 제품의 이름을 찾는다. 이것은 제품의 이름을 돌려준다.

이제는 실례를 확장하여 실제 계산을 수행하는 계산마당들을 포함할수 있다.

머리부파일 sql/overview/subclass4/main.h는 앞의 실례로부터 변경되지 않지만 구성자와 calculateField()함수는 간단한 확장을 요구한다. 매개를 차례로 고찰한다.

```
// sql/overview/subclass4/main.cpp로부터
  InvoiceItemCursor::InvoiceItemCursor() :
QSqlCursor( "invoiceitem" )
  {
    QSqlFieldInfo productName( "productname", QVariant::String );
    append( productName );
    setCalculated( productName.name(), TRUE );

    QSqlFieldInfo productPrice( "price", QVariant::Double );
    append( productPrice );
    setCalculated( productPrice.name(), TRUE );

    QSqlFieldInfo productCost( "cost", QVariant::Double );
    append( productCost );
    setCalculated( productCost.name(), TRUE );
  }
```

두개의 특수마당 price와 cost를 창조하여 유표의 마당모임에 추가한다. 둘다 setCalculated()호출에 의하여 계산마당에 등록된다.

```
// sql/overview/subclass4/main.cpp로부터
  QVariant InvoiceItemCursor::calculateField( const QString & name )
  {

    if ( name == "productname" ) {
      QSqlQuery query( "SELECT name FROM prices WHERE id=" +
```

```
    field( "pricesid" )->value().toString() );
      if ( query.next() )
        return query.value( 0 );
    }
    else if ( name == "price" ) {
      QSqlQuery query( "SELECT price FROM prices WHERE id=" +
    field( "pricesid" )->value().toString() );
      if ( query.next() )
        return query.value( 0 );
    }
    else if ( name == "cost" ) {
      QSqlQuery query( "SELECT price FROM prices WHERE id=" +
    field( "pricesid" )->value().toString() );
      if ( query.next() )
        return QVariant( query.value( 0 ).toDouble() *
        value( "quantity").toDouble() );
    }

    return QVariant( QString::null );
  }
```

calculateField()함수는 현재 3개의 서로 다른 마당들의 값을 계산해야 하므로 현저하게 확장되였다. productname과 price마당들은 prices표에서 pricesid를 열쇠로 하는 대응하는 값들을 탐색하여 생성한다. cost마당은 price에 quantity을 곱하여 간단히 계산할수 있다. 단가는 calculateField()가 돌려주어야 할 형인 QVariant로 강제변환된다.

매개 계산마당이 각이한 표나 보기에 대한 검색으로 될수 있는 실제응용프로그람과 비슷한 실례로 만들기 위하여 하나가 아니라 3개의 개별적인 질문을 썼다.

추가해야 할 마지막 기능은 사용자가 새 레코드를 삽입하려고 시도할 때의 기정값들이다.

```
// sql/overview/subclass5/main.h로부터
      QSqlRecord *primeInsert();
```

자체의 primeInsert()함수를 재정의하기 위하여 이 함수를 선언한다.

구성자와 calculateField()함수는 변경되지 않는다.

```
// sql/overview/subclass5/main.cpp로부터
  QSqlRecord *InvoiceItemCursor::primeInsert()
  {
    QSqlRecord *buffer = editBuffer();
    QSqlQuery query( "SELECT NEXTVAL( 'invoiceitem_seq' )" );
    if ( query.next() )
      buffer->setValue( "id", query.value( 0 ) );
    buffer->setValue( "paiddate", QDate::currentDate() );
    buffer->setValue( "quantity", 1 );

    return buffer;
  }
```

유표가 삽입과 갱신에 사용되는 내부편집완충기의 지적자를 얻는다. id마당은

invoiceitem_seq에 의해 생성되는 유일한 옹근수이다. paiddate마당의 기정값을 오늘의 날자로 설정하고 quantity의 기정값을 1로 설정한다. 끝으로 완충기의 지적자를 돌려준다. 나머지 코드는 이전판으로부터 변경되는것이 없다.

8. 실례표들

사용된 실례표들은 다음의 표준SQL에 의해 다시 창조될수 있다. 자기의 특정한 자료기지에 사용하는것과 일치하도록 SQL을 수정할 필요가 있다.

```
create table people (id integer primary key, name char(40))
create table staff (id integer primary key, forename char(40),
    surname char(40), salary float, statusid integer)
create table status (id integer primary key, name char(30))
create table creditors (id integer primary key, forename char(40),
   surname char(40), city char(30))
create table prices (id integer primary key, name char(40), price float)
create table invoiceitem (id integer primary key,
    pricesid integer, quantity integer, paiddate date)
```

우의 calculateField()실례에서는 순서렬을 리용하였다. 순서렬은 모든 자료기지들에서 유지되는것은 아니라는것을 알아야 한다.

```
create sequence invoiceitem_seq
```

9. 지원하는 구동프로그람들

SQL모듈은 각이한 자료기지API들과 교제하기 위하여 구동프로그람플라그인을 사용한다. SQL모듈API가 자료기지에 의존하지 않으므로 모든 자료기지에 고유한 코드는 이 구동프로그람들내에 포함된다. 일부 구동프로그람들은 Qt와 함께 공급되고 다른 일부는 추가될수 있다. 구동프로그람원천코드가 공급되고 자체의 구동프로그람쓰기를 위한 모형으로 사용될수 있다.

알아두기: 구동프로그람플라그인을 구축하기 위하여 자기의 자료기지관리체계용의 적당한 의뢰기서고를 가지고있어야 한다. 이것은 DBMS에 의해 공개된 API에로의 접근을 제공한다. 대다수 설치프로그람들도 사용자의 요구에 따라 《개발서고》를 설치한다. 이 서고들은 DBMS와의 저수준통신을 보장할수 있다.

Qt에 적재되는 구동프로그람들은 다음과 같다.

- QDB2 – IBM DB2구동프로그람 (v7.1이상)
- QIBASE – Borland Interbase 구동프로그람
- QMYSQL3 – MySQL 구동프로그람
- QOCI8 – Oracle Call Interface 구동프로그람, 8, 9 및 10판
- QODBC3 – Open Database Connectivity 구동프로그람
- QPSQL7 – PostgreSQL v6.x and v7.x 구동프로그람
- QSQLITE – SQLite 구동프로그람
- QTDS7 – Sybase Adaptive Server
- GPL허가의 비호환성으로인하여 Qt공개원천판으로 모두 적재되는것은 아니다.

1) configure를 리용한 구동프로그람들의 구축

Qt configure스크립트는 자동적으로 자기 콤퓨터에서 유효의뢰기서고들을 탐지한다. 어떤 구동프로그람들을 구성할수 있는가 알려면 "configure -help"를 실행한다. 다음과 비슷한 출력이 얻어진다.

```
Possible values for <driver>: [ mysql oci odbc psql tds ]
Auto-Detected on this system: [ mysql psql ]
```

Windows에서는 configure스크립트가 자동탐지되지 않는다.

configure스크립트는 필요한 서고와 머리부파일들이 표준경로에 없으면 탐지할수 없으므로 "-I"과 "-L"스위치들을 리용하여 이 경로들을 지정할 필요가 있다. 실례로 자기의 MySQL머리부파일들은 /usr/local/mysql에 (혹은 C:\mysql\include configure: -I/usr/local/mysql (혹은 -I C:\mysql\include Windows).

Windows에서 -I파라메터는 파일이름에서 공백을 받아들이지 않으므로 8.3이름형식을 리용한다. 즉 C:\progra~1\mysql을 C:\program files\mysql대신에 리용한다.

자료기지구동프로그람을 자기의 Qt서고에 정적으로 구축하는데 -qt-sql-<구동프로그람> 파라메터를 리용하고 구동프로그람을 플라그인으로서 구축하는데 -plugin-sql-<구동프로그람> 을 리용한다.

2) 수동적인 플라그인구축

(1) QMYSQL3 - MySQL 3.x과 MySQL 4.x

- 일반정보

MySQL 3.x는 기정으로 SQL일괄처리를 유지하지 않는다. 이 기능을 제공하는 backend들이 있다. MySQL의뢰기서고들의 최근판(>3.23.34)은 그 수정된 봉사기들에서 일갈처리를 리용하게 한다.

최근의 의뢰기서고를 가지고있고 일괄처리가능한 MySQL봉사기에 련결한다면 QSqlDriver::hasFeature(QSqlDriver::Transactions)함수호출은 TRUE를 돌려주고 SQL일괄처리를 사용할수 있다.

플라그인이 MySQL 4.x의뢰기서고들에 대하여 콤파일되면 일갈처리는 기정으로 가능해진다.

http://www.mysql.com에서 MySQL에 대한 정보를 찾을수 있다.

- Unix/Linux에서 플라그인구축방법

MySQL머리부파일들가 공유서고 libmysqlclient.so를 요구한다. 자기의 Linux배포물에 따라서 보통 "mysql-devel"이라고 부르는 패키지를 설치해야 한다.

qmake는 MySQL머리부파일과 공유서고들을 어디서 찾는가 말하고(여기서는 MySQL이 /usr/local에 설치되여있다고 가정한다) make를 실행한다.

```
cd $QTDIR/plugins/src/sqldrivers/mysql
qmake -o Makefile "INCLUDEPATH+=/usr/local/include" "LIBS+=-L/usr/local/lib -lmysqlclient" mysql.pro
make
```

- Windows에서 플라그인구축방법

MySQL설치파일들을 얻어야 한다. SETUP.EXE를 실행하고 "Custom Install"을 선택한다. "Libs & Include Files"모듈을 설치한다. 프라그인을 다음과 같이 구축한다. (여기서는 MySQL이 C:\MYSQL에 설치된다고 가정한다.)

```
cd %QTDIR%\plugins\src\sql구동프로그람s\mysql
qmake -o Makefile "INCLUDEPATH+=C:\MYSQL\INCLUDE" "LIBS+=C:\MYSQL\LIB\OPT\LIBMYSQL.LIB" mysql.pro
nmake
```

Microsoft콤파일러를 리용하지 않는다면 우의 문에서 nmake를 make 로 교체한다.

(2) QOCI8 - Oracle호출대면부 (OCI)

- 일반정보

Qt OCI플라그인은 Oracle 8과 Oracle 9을 둘다 유지한다. Oracle봉사기에 련결한 다음 플라그인은 자료기지판과 가능한 특성들을 자동탐지한다.

- 유니코드지원

Oracle봉사기가 유니코드를 유지하면 OCI플라그인은 UTF-8부호화를 리용하여 봉사기와 교제한다.

- BLOB/LOB지원

BLOB(Binary Large Object)를 읽고 쓸수 있으나 이 과정이 많은 기억기를 요구한다는것을 알아야 한다.

Oracle 9는 LOB렬들을 가지는 흘림가능결과모임을 유지하지 않으며 정방향질문만 리용하여 LOB마당을 선택할수 있다(QSqlQuery::setForwardOnly()참고).

BLOB삽입은 BLOB가 대리기호로 제한되는 준비된 질문 혹은 내적으로 이것을 수행하는 준비된 질문을 리용하는 QSqlCursor를 리용하여 수행되여야 한다($QTDIR/examples/sql/blob참고).

- Unix/Linux에서 플라그인구축방법

구동프로그람구축에 필요한 모든 파일들은 표준Oracle의뢰기를 적재해야 한다.

Oracle서고파일들이 구동프로그람구축에 필요하다.

- libclntsh.so (모든 판)
- libwtc8.so (Oracle 8만) 혹은 libwtc9.so (Oracle 9만)

qmake는 Oracle 머리부파일과 공유서고들을 어디서 찾는가 말하고(변수 $ORACLE_HOME가 Oracle이 설치되는 등록부를 가리킨다고 가정한다) make를 실행한다.

Oracle 8을 리용하고있다면

```
cd $QTDIR/plugins/src/sql구동프로그람s/oci
qmake -o Makefile "INCLUDEPATH+=$ORACLE_HOME/rdbms/public $ORACLE_HOME/rdbms/demo" "LIBS+=-L$ORACLE_HOME/lib -lclntsh -lwtc8" oci.pro
make
```

Oracle 9를 리용한다면

```
cd $QTDIR/plugins/src/sql구동프로그람s/oci
qmake -o Makefile "INCLUDEPATH+=$ORACLE_HOME/rdbms/public $ORACLE_HOME/rdbms/demo" "LIBS+=-L$ORACLE_HOME/lib -lclntsh -lwtc9" oci.pro
make
```

Oracle 10을 사용한다면

```
cd $QTDIR/plugins/src/sql구동프로그람s/oci
qmake -o Makefile "INCLUDEPATH+=$ORACLE_HOME/rdbms/public $ORACLE_HOME/rdbms/demo" "LIBS+=-L$ORACLE_HOME/lib -lclntsh" oci.pro
make
```

일부 판의 OCI의뢰기서고들은 완료시에 이 서고에 련결된 프로그람들은 segmentation fault오유를 일으킨다. 이것은 QOCI8구동프로그람이 플라그인으로서 콤파일되면 발생한다. 이 문제에 대하여 작업하기 위하여 Qt서고자체로 구동프로그람을 콤파일하거나 선택 '-DQT_NO_LIBRARY_UNLOAD'를 가지고 Qt를 구성한다. Oracle 9에서는 "LIBS+=$ORACLE_HOME/lib/libclntst9.a"를 리용하여 정적OCI서고에 련결할수 있다.

- Windows에서 플라그인구축방법

Oracle의뢰기설치CD로부터 Oracle의뢰기설치프로그람에서 "Programmer"를 선택하면 플라그인구축에 충분하다.

플라그인을 다음과 같이 구축한다. (여기서 Oracle의뢰기가 C:\oracle에 설치된다는것을 가정한다.)

set INCLUDE=%INCLUDE%;c:\oracle\oci\include
set LIB=%LIB%;c:\oracle\oci\lib\msvc
cd %QTDIR%\plugins\src\sql구동프로그람s\oci
qmake -o Makefile oci.pro
nmake

자기 응용프로그람을 실행할 때 자기의 PATH 환경변수에 oci.dll경로를 추가해야 한다.

set PATH=%PATH%;c:\oracle\bin

Microsoft콤파일러를 사용하지 않는다면 우의 문에서 nmake를 make로 교체한다.

(3) QODBC3 - Open Database Connectivity (ODBC)

- 일반정보

ODBC는 공통대면부를 리용하여 여러 DBMS에 련결하게 하는 일반대면부이다. QODBC3구동프로그람은 ODBC구동프로그람관리기에 련결하고 유효자료원천들을 호출하게 한다. 또한 자기 체계에 설치되는 ODBC 구동프로그람 관리기용의 ODBC 구동프로그람들을 설치하고 환경을 구성해야 한다. 그때 QODBC3플라그인은 자기의 Qt프로젝트에서 이 자료원천들을 사용하게 한다.

Windows 95이후의 체계들에서 ODBC 구동프로그람관리기는 기정으로 설치되여야 하고 Unix체계에서 우선 설치하여야 할 실현이 있다. 자기 응용프로그람을 사용하는 각 의뢰기는 설치된 ODBC구동프로그람관리기가 있어야 하며 그렇지 않으면 QODBC3 플라그인은 동작하지 않는다.

ODBC 자료원천에 련결할 때 실제자료기지이름이 아니라 ODBC자료원천의 이름을 QSqlDatabase::setDatabaseName()함수에 넘겨야 한다.

QODBC3플라그인은 ODBC호환구동프로그람 관리기2.0이후에서 작업할것을 요구한다. 일부 ODBC 구동프로그람들은 2.0판호환을 요구하지만 필요한 모든 기능을 제공하지 않는다. 그러므로 QODBC3플라그인은 자료원천을 련결이 확립된 후에 사용할수 있는가 검사하고 검사가 실패하면 작업을 거부한다. 이렇게 동작하지 않는다면 파일 qsql_odbc.cpp에서 #define ODBC_CHECK_DRIVER행을 삭제할수 있다.

ODBC자료원천의 아주 느린 호출을 보았다면 ODBC호출추적이 ODBC자료원천 관리기에서 차단된다는것을 확인한다.

- 유니코드지원

QODBC3플라그인은 UNICODE가 정의되면 유니코드 API를 사용한다. Windows NT기반체계에서 이것은 기정이다. ODBC구동프로그람과 DBMS가 물론 유니코드를 유지해야 한다.

Oracle 9 ODBC구동프로그람(Windows)에 대하여 ODBC 구동프로그람 관리기에서 "SQL_WCHAR support"을 검사할 필요가 있으며 그렇지 않으면 Oracle은 모든 유니코드문자렬을 국부8 bit로 변환한다.

- Unix/Linux에서 플라그인구축방법

unixODBC를 리용할것을 권고한다. http://www.unixodbc.org에서 최근판과 ODBC 구동프로그람들을 찾을수 있다. unixODBC머리부파일과 공유서고들을 요구한다.

qmake에게 어디서 unixODBC머리부파일과 공유서고들을 찾는가를 알리고(여기서는 unixODBC가 /usr/local/unixODBC에 설치된다고 가정) make를 실행한다.

cd $QTDIR/plugins/src/sqldrivers/odbc
qmake "INCLUDEPATH+=/usr/local/unixODBC/include" "LIBS+=-L/usr/local/unixODBC/lib -lodbc"
make

- Windows에서 플라그인구축방법

ODBC머리부파일들은 이미 정확한 등록부들에 설치되여있어야 한다. 다음과 같이

플라그인을 구축해야 한다.

```
cd %QTDIR%\plugins\src\sql구동프로그람s\odbc
qmake -o Makefile odbc.pro
nmake
```

Microsoft콤파일러를 사용하지 않는다면 우의 문에서 nmake를 make로 교체한다.

(4) QPSQL7 - PostgreSQL 6판과 7판

- 일반정보

QPSQL7구동프로그람은 PostgreSQL 6판과 7판을 둘다 유지한다. 최근판의 PostgreSQL의뢰기서고(libpq)가 안전하고 역호환되므로 이것을 가지고 콤파일할것을 권고한다.

6판에서 실은 libpq에 플라그인을 련결하려고 한다면 PostgreSQL 6.5.3과 같은 현재판을 권고하며 그렇지 않으면 7판의 봉사기에 대한 련결이 동작하지 않는다.

구동프로그람은련결이 성공한 후에 PostgreSQL의 봉사기판을 자동탐지한다. 봉사기가 너무 낡거나 판정보를 결정할수 없으면 경고가 발생된다.

- 유니코드지원

QPSQL7구동프로그람은 련결하고있는 PostgreSQL자료기지가 유니코드를 지원하는가 안하는가를 자동적으로 탐지한다. 유니코드는 봉사기가 그것을 지원한다면 자동적으로 사용된다. 구동프로그람은 오직 UTF-8부호화만 유지한다. 자기의 자료기지가 다른 부호화를 사용한다면 봉사기는 유니코드변환을 유지하여 콤파일되여야 한다.

유니코드유지는 PostgreSQL 7.1판에서 도입되였으며 봉사기와 의뢰기의 두 서고가 여러바이트를 유지하여 콤파일되였으면 동작한다.(PostgreSQL봉사기를 여러바이트가 가능하게 설정하는 방법에 대한 자세한 정보는 《PostgreSQL Administrator Guide》 제5장을 보시오.)

- BLOB지원

대규모2진객체(Binary Large Object)는 PostgreSQL 7.1판이상에서 BYTEA 마당을 통하여 지원된다. OID형의 마당들은 읽어들일수 있으나 써넣기할수 없다. PostgreSQL지령lo_import를 리용하여 OID마당들에 2진자료를 삽입한다.

- Unix/Linux에서 플라그인구축방법

pq의뢰기서고와 대응하는 머리부파일들을 설치하는것으로 충분하지 않다. PostgreSQL 원천배포물을 얻어서 환경구성스크립트를 실행해야 한다. 2진배포물을 이미 설치하였다면 그것을 구축할 필요가 없다. 원천배포물은 QPSQL7플라그인이 보통 2진배포물의 부분이 아닌 한쌍의 머리부파일들에 기초하므로 필요된다.

qmake가 PostgreSQL머리부파일과 공유서고들을 찾게 하기 위하여 qmake가 다음과 같은 방법으로 실행한다. (PostgreSQL원천은 /usr/src/psql에서 찾을수 있다는것을 전제로 한다.)

```
cd $QTDIR/plugins/src/sqldrivers/psql
qmake -o Makefile "INCLUDEPATH+=/usr/src/psql/src/include /usr/src/psql/src/interfaces/libpq" "LIBS+=-L/usr/lib -lpq" psql.pro
make
```

- Windows에서 플라그인구축방법

PostgreSQL문서에서 설명하는것처럼 PostgreSQL원천배포물을 풀고 구축한다. PostgreSQL원천이 C:\psql에 상주한다는것을 전제로 하고 플라그인을 다음과 같이 구축한다.

```
cd %QTDIR%\plugins\src\sql구동프로그람s\psql
qmake -o Makefile "INCLUDEPATH+=C:\psql\src\include C:\psql\src\interfaces\libpq" psql.pro
```

nmake
libpq.dll서고에로의 경로를 자기의 PATH환경변수에 추가하여 Windows가 그것을 찾을수 있도록 해야 한다. 이 경우에 C:\psql\src\interfaces\libpq\Release. Microsoft콤파일러를 사용하지 않는다면 우의 문에서 nmake를 make로 교체한다.
(5) QTDS7 - Sybase Adaptive Server
- Unix/Linux에서 플라그인구축방법
Unix하에서 TDS통신규약을 지원하는 두 서고를 사용할수 있다.
- FreeTDS, TDS 통신규약(http://www.freetds.org)의 자유실현. FreeTDS를 아직 안정하지 않으므로 어떤 기능은 기대한대로 동작하지 않을수 있다.
- Sybase공개의뢰기, http://www.sybase.com로부터 얻을수 있다. Linux사용자들은 http://linux.sybase.com로부터 공개의뢰기RPM을 얻을수 있다.
사용하는 서고에 관계없이 공유목적파일libsybdb.so이 필요하다. SYBASE환경변수가 의뢰기서고를 설치한 등록부를 가리키게 설정하고 qmake를 실행한다.
cd $QTDIR/plugins/src/sqldrivers/tds
qmake -o Makefile "INCLUDEPATH=$SYBASE/include" "LIBS=-L$SYBASE/lib -lsybdb"
make
- Windows에서 플라그인구축방법
Microsoft나 Sybase공개의뢰기(http://www.sybase.com)에 의해 제공되는 DB-Library를 사용할수 있다. 플라그인을 구축하려면 NTWDBLIB.LIB을 포함해야 한다.
cd %QTDIR%\plugins\src\sql구동프로그람s\tds
qmake -o Makefile "LIBS+=NTWDBLIB.LIB" tds.pro
nmake
기정으로 Microsoft서고는 Windows에서 사용되므로 Sybase공개의뢰기사용을 요구하면 %QTDIR%\src\sql\drivers\tds\qsql_tds.cpp에서 Q_USE_SYBASE를 정의해야 한다.
(6) QDB2 - IBM DB2구동프로그람 (v7.1이상)
- 일반정보
Qt DB2플라그인은 IBM DB2자료기지들을 호출할수 있게 한다. IBM DB2 v7.1과 7.2를 가지고 시험하였다. QDB2플라그인의 콤파일에 필요한 머리부와 서고파일들을 포함하는 IBM DB2개발의뢰기서고를 설치해야 한다.
QDB2구동프로그람은 준비되여있는 질문, 유니코드문자렬의 읽고 쓰기 및 BLOB의 읽고 쓰기를 지원한다.
DB2에 보관된 수속들을 호출할 때 정방향의 질문만 사용할것을 제안한다(QSqlQuery::setForwardOnly()참고).
- Unix/Linux에서 플라그인구축방법
cd $QTDIR/plugins/src/sql구동프로그람s/db2
qmake -o Makefile "INCLUDEPATH+=$DB2DIR/include" "LIBS+=-L$DB2DIR/lib -ldb2"
make
- Windows에서 플라그인구축방법
DB2머리부파일들은 이미 정확한 등록부에 설치되여야 한다. 플라그인을 다음과 같이 구축해야 한다.
cd %QTDIR%\plugins\src\sqldrivers\db2
qmake -o Makefile "INCLUDEPATH+=<DB2 home>/sqllib/include" "LIBS+=<DB2 home>/sqllib/lib/db2cli.lib"
nmake

Microsoft콤파일러를 사용하지 않는다면 우의 문에서 nmake를 make로 교체한다.

(7) QSQLITE - SQLite구동프로그람

Qt SQLite플라그인은 SQLite자료기지를 호출할수 있게 한다. SQLite는 프로쎄스내자료기지이다. 이것은 자료기지봉사기를 가질 필요가 없다는것을 의미한다. SQLite는 련결을 열 때 자료기지이름으로서 설정되여야 하는 하나의 파일에 조작한다. 그 파일이 존재하지 않으면 SQLite는 그것을 창조하려고 한다. 또한 SQLite는 기억기내자료기지들을 지원하며 단지 자료기지이름으로서 ":memory:"를 넘긴다.

SQLite는 다중사용자들과 다중일괄처리에 관한 제한이 있다. 각이한 일괄처리에 대하여 읽기와 쓰기를 하려면 자기 응용프로그람은 하나의 일괄처리를 위임하거나 역전될 때까지 해방할수 있다.

SQLite는 형에 대한 지원이 없으며 매개 값은 문자자료로서 취급된다. 그러므로 BLOB는 유지되지 않는다.

http://www.sqlite.org에서 SQLite에 대한 정보를 찾을수 있다.

SQLite는 Qt에서 제3자서고로서 적재된다. configure스크립트에 다음의 파라메터들을 넘기여 구축한다. 즉 -plugin-sql-sqlite (플라그인로서) 혹은 -qt-sql-sqlite (Qt서고에로 직접 련결).

Qt에 적재된 SQLite서고를 사용하지 않으면 수동적으로 구축할수 있다. ($SQLITE를 SQLite가 상주하는 등록부로 바꾼다.)

```
cd $QTDIR/plugins/src/sqldrivers/sqlite
qmake -o Makefile "INCLUDEPATH+=$SQLITE/include" "LIBS+=-L$SQLITE/lib -lsqlite"
make
```

(8) QIBASE - Borland Interbase구동프로그람

- 일반정보

Qt Interbase플라그인은 Interbase와 Firebird자료기지의 호출을 간단하게 한다. Interbase는 의뢰기-봉사기로서 사용되거나 국부파일들에 조작하는 봉사기없이 사용될수 있다. 자료기지파일은 련결이 이루어지기 전에 존재해야 한다.

Interbase는 자료기지파일에로의 완전경로를 지정할것을 요구하며 그것이 국부적으로 혹은 다른 봉사기에 보관되여있으면 문제가 생기지 않는다.

```
myDatabase->setHostName("MyServer");
myDatabase->setDatabaseName("C:\\test.gdb");
```

이 플라그인을 구축하려면 Interbase/Firebird개발머리부와 서고들이 요구된다.

GPL로 인하여 Qt공개원천판의 사용자들은 이 플라그인을 Interbase의 상업판에 련결하는것이 허용되지 않는다. Firebird를 사용하거나 무료판의 Interbase을 사용하시오.

- Unix/Linux에서 플라그인구축방법

다음은 Interbase 혹은 Firebird가 /opt/interbase에 설치된것을 전제로 한다.

```
cd $QTDIR/plugins/src/sql구동프로그람s/ibase
qmake -o Makefile "INCLUDEPATH+=/opt/interbase/include" "LIBS+=-L/opt/interbase/lib" ibase.pro
make
```

- Windows에서 플라그인구축방법

다음것은 Interbase나 Firebird가 C:\interbase에 설치된것을 전제로 한다.

```
cd %QTDIR%\plugins\src\sql구동프로그람s\ibase
qmake -o Makefile "INCLUDEPATH+=C:\interbase\include" ibase.pro
nmake
```

Microsoft콤파일러를 사용하지 않는다면 우의 문에서 nmake를 make로 교체한다.

C:\interbase\bin가 PATH에 있어야 한다.

3) 주의할 점

자기 프로젝트에 사용하고있는 콤파일러로 콤파일한 의뢰기서고를 항상 리용하여야 한다. 의뢰기서고들을 콤파일하기 위한 원천배포물을 얻을수 없다면 사전콤파일된 서고가 자기 콤파일러와 호환성있는가 확인하여 하며 그렇지 않으면 많은 "undefined symbols"오유를 얻을수 있다. 일부 콤파일러는 서고를 변한하기 위한 도구들을 가지고 있다. 실례로 Borland는 Microsoft Visual C++로 생성한 서고들을 변환하기 위한 도구 COFF2OMF.EXE를 적재한다.

플라그인의 콤파일에서 성공하였지만 적재할수 없으면 다음의 요구가 만족되는가 확인한다.

- 공유Qt서고를 리용하고있는가 확인한다. 플라그인들을 정적구축에 사용할수 없다.
- 환경변수 QTDIR가 정확한 등록부를 가리키는가 확인한다. $QTDIR/plugins/sqldrivers등록부로 가서 그 등록부에 플라그인이 존재하는가 확인한다.
- DBMS의 의뢰기서고가 체계에서 사용할수 있는가 확인한다. Unix에서 지령ldd을 실행하고 파라메터로서 플라그인이름을 넘긴다. 실례로 ldd libqsqlmysql.so. 임의의 의뢰기서고를 찾을수 없으면 경고를 얻을수 있다. Windows에서 Visual Studio의 의존관계추적기를 사용할수 있다.

플라그인적재와 관련한 문제들이 발생하면 다음과 같은 출력을 볼수 있다.

QSqlDatabase warning: QMYSQL3 구동프로그람 not loaded

QSqlDatabase: available 구동프로그람s: QMYSQL3

문제는 플라그인이 틀린구축건이 있는것이다. 오유수정목적에서는 $HOME/.qt/qt_plugins_(qtversion).rc 파일에 대응하는 항목을 삭제한다.

이 플라그인을 다음에 적재할 때 더 자세한 오유통보를 줄것이다.

4) 자체의 자료기지구동프로그람을 쓰는 방법

QSqlDatabase는 자료기지 구동프로그람 플라그인들의 적재와 관리에 응답할수 있다. 자료기지가 추가되면(QSqlDatabase::addDatabase()참고) 적당한 구동프로그람 플라그인이 적재된다(QSqlDriverPlugin리용). QSqlDatabase는 구동프로그람 플라그인에 기초하여 qsqldriver.html과 qsqlresult.html의 대변부를 제공한다.

QSqlDriver는 SQL자료기지구동프로그람의 기능을 정의하는 추상기초크라스이다. 이것은 QSqlDriver::open()과 QSqlDriver::close()과 같은 함수들을 포함한다. QSqlDriver는 자료기지련결에 응답하고 적당한 환경을 확립한다. 또한 QSqlDriver는 특별한 자료기지 API에 적합한 QSqlQuery객체들을 창조할수 있다. QSqlDatabase는 구체적인 실현을 제공하는 QSqlDriver에로의 직접함수호출을 촉진시킨다.

QSqlResult는 SQL자료기지질문의 기능을 정의하는 추상기초크라스이다. 이것은 SELECT, UPDATE, 및 ALTER TABLE과 같은 문들을 포함한다. QSqlResult는 QSqlResult::next(), QSqlResult::value()와 같은 함수들을 포함한다. QSqlResult는 자료기지에 질문보내기, 결과자료 돌려주기 등에 응답할수 있다. QSqlQuery는 구체적인 실현을 제공하는 "qsqlresult.html"에로의 수많은 직접호출을 촉진시킨다.

QSqlDriver와 QSqlResult는 밀접히 결합된다. Qt SQL 구동프로그람을 실현할 때 두 클라스는 파생클라스화되여야 하고 각 클라스에서 추상가상메쏘드들이 실현되여야 한다.

Qt SQL구동프로그람을 플라그인으로 실현하기 위하여(그것이 실행시에 Qt서고를 인식하고 적재하도록 하기 위하여) 구동프로그람은 Q_EXPORT_PLUGIN 마크로를 사용해야 한다. (자세한 정보는 plugins-howto.html을 읽으시오.) 또한 QTDIR/plugins/src/sqldrivers와 QTDIR/src/sql/drivers에서 Qt에 제공되는 SQL플라그인들에

서 이것을 수행하는 방법을 검사한다.

다음의 코드는 SQL구동프로그람의 골격으로 사용될수 있다.

```
class QNullResult : public QSqlResult
{
public:
    QNullResult( const QSqlDriver* d ): QSqlResult( d ) {}
    ~QNullResult() {}
protected:
    QVariant    data( int ) { return QVariant(); }
    bool        reset ( const QString& ) { return FALSE; }
    bool        fetch( int ) { return FALSE; }
    bool        fetchFirst() { return FALSE; }
    bool        fetchLast() { return FALSE; }
    bool        isNull( int ) { return FALSE; }
    QSqlRecord  record() { return QSqlRecord(); }
    int         size()  { return 0; }
    int         numRowsAffected() { return 0; }
};

class QNullDriver: public QSqlDriver
{
public:
    QNullDriver(): QSqlDriver() {}
    ~QNullDriver() {}
    bool    hasFeature( DriverFeature ) const { return FALSE; }
    bool    open( const QString&, const QString&, const QString&,
                  const QString&, int )
            { return FALSE; }
    void    close() {}
    QSqlQuery createQuery() const { return QSqlQuery( new
QNullResult( this ) ); }
};
```

제6절. 표모듈

표모듈은 유연하고 편집가능한 표창문부품 QTable을 제공한다. 많은 응용프로그람들에서 QTable은 직접 간단히 사용될수 있으며 살창모양의 편집가능세포들을 제공한다. 또한 QTable은 간단한 방법으로 파생클라스화되여 대규모의 보기드문 표, 실례로 수백만개의 세포를 가지는 표를 제공한다.

그림 6-3. 표모듈

표모듈은 다음과 같은 클라스들을 제공한다.

• QTable자체는 표계산자료나 자료기지자료와 같은 표형식자료를 현시하고 편집할 능력을 사용자에게 제공해야 할 때마다 선택하는 창문부품이다.
• QTableItem객체들은 세포의 내용을 보유하는 표의 매개 항목의 자료들로 QTable을 채우는데 쓰인다.
• QComboTableItem클라스는 QTable들에 기억효과 복합칸항목들을 제공한다.
• QCheckTableItem클라스는 QTable들에 기억효과 검사칸항목들을 제공한다.
• QTableSelection은QTable안의 세포선택에 대한 접근을 제공한다.
• QHeader는 표의 수평제목(렬제목)과 수직제목(행제목)에 대한 접근을 제공한다.

제7절. 작업공간모듈

작업공간모듈은 다중문서대면부(MDI)용의 장식된 문서창문들을 포함할수 있는 작업공간창문을 제공한다.

이것은 하나의 클라스 QWorkspace에 의해 실현된다.

제8절. XML모듈

1. Qt에서 XML구성방식

XML모듈은 SAX2(Simple API for XML)을 리용하는 잘 형식화된 XML문법해석기와 DOM Level 2 (Document Object Model)의 실현을 제공한다.

SAX는 XML문법해석기용 사건기초표준대면부이다. Qt대면부는 SAX2 Java실현을 따른다. 그 명명구조는 Qt명명관례에 맞게 채용되였다.

SAX2려파기와 읽기기구공장의 지원은 개발중에 있다. Qt실현은 Java대면부에서 제시된 SAX1호환클라스들을 포함하지 않는다.

DOM Level 2는 XML문서의 요소들을 나무구조로 넘기는 XML대면부용 W3C권고이다.

Qt는 표 6-3과 같은 XML관련클라스들을 제공한다.

표 6-3. XML관련클라스

클라스	간단한 설명
QDomAttr	QDomElement 의 한개 속성을 표시한다.
QDomCDATASection	XML CDATA절을 표시한다.
QDomCharacterData	DOM안의 일반문자렬을 표시한다.
QDomComment	XML설명문을 표시한다.
QDomDocument	XML문서의 표시
QDomDocumentFragment	보통 완전한 QDomDocument이 아닌 QDomNode들의 나무
QDomDocumentType	문서나무에서 DTD의 표시
QDomElement	DOM나무의 한 요소를 표시한다.
QDomEntity	XML실체를 표시한다.
QDomEntityReference	XML실체참고를 표시한다.
QDomImplementation	DOM실현의 특성에 대한 정보
QDomNamedNodeMap	이름에 의해 호출할수 있는 마디집합
QDomNode	DOM나무의 모든 마디들의 기초클라스
QDomNodeList	QDomNode객체들의 목록
QDomNotation	XML표기법을 표시한다.
QDomProcessingInstruction	XML처리지령을 표시한다.
QDomText	문법해석된 XML문서의 본문자료를 표시한다.
QXmlAttributes	XML속성
QXmlContentHandler	XML자료의 론리적내용을 알리기 위한 대면부
QXmlDeclHandler	XML자료의 선언내용을 알리기 위한 대면부
QXmlDefaultHandler	모든 XML처리함수클라스들의 기정실현
QXmlDTDHandler	XML자료의 DTD내용을 알리기 위한 대면부
QXmlEntityResolver	XML자료에 포함된 외부실체를 해결하기 위한 대면부
QXmlErrorHandler	XML자료로서 오유를 알리기 위한 대면부
QXmlInputSource	QxmlReader파생클라스용 입력자료
QXmlLexicalHandler	XML자료의 어휘적내용을 알리기 위한 대면부
QXmlLocator	파일안에서 문법해석위치에 대한 정보를 가지는 XML처리함수 클라스
QXmlNamespaceSupport	이름공간유지를 포함하려고 하는 XML읽기기구용 방조클라스
QXmlParseException	QXmlErrorHandler대면부에서 오유통보에 사용한다.
QXmlReader	XML읽기기구(SAX2문법해석기용)의 대면부
QXmlSimpleReader	간단한 XML읽기기구(SAX2 문법해석기용)의 실현

2. Qt SAX2클라스들

1) SAX2

SAX2대면부는 사건구동형기구로서 문서정보를 사용자에게 제공한다. 이 문맥에서 《사건》은 문법해석기에 의해 통보되는것을 의미한다. 례를 들면 문법해석기는 시작꼬리표나 끝꼬리표 등과 만나게 된다.

그것을 상상하기 위해 다음의 실례를 고찰한다.

<quote>A quotation.</quote>

우의 문서를 읽을 때(SAX2문법해석기는 보통 《읽기기구》로서 서술된다) 3개의 사건들이 발생한다.

① 시작꼬리표가 발생한다(<quote>).

② 문자자료(즉 본문)가 발견된다("A quotation.").

③ 끝꼬리표가 해석된다(</quote>).

그러한 사건이 발생할 때마다 문법해석기가 그것을 알리고 사건처리함수들을 설정하여 이 사건들에 응답한다.

이것은 XML문서를 읽어들이는 단순한 고속수법이지만 자료를 보관하지 않고 단순히 처리하고 계렬적으로 버리므로 조작이 복잡해진다. DOM대면부는 전체문서를 나무구조로 읽고 보관하며 이것은 기억기를 많이 소비하지만 문서의 구조에 조작하기 쉽게 한다.

Qt XML모듈은 추상클라스 QXmlReader를 제공하며 이 클라스는 잠재적인 SAX2읽기기구용 대면부를 정의한다. Qt는 읽기기구실현 QXmlSimpleReader를 포함하며 이것은 파생클라스화를 받아들이기 쉽다.

읽기기구는 특수처리함수클라스들을 통하여 사건들의 해석정형을 알린다(표 6-4).

표 6-4. 특수처리함수클라스

처리함수클라스	설명
QXmlContentHandler	문서의 내용(실례로 시작꼬리표나 문자)과 관련한 사건들을 보고한다.
QXmlDTDHandler	DTD(실례로 표기선언)와 관련한 사건들을 알린다.
QXmlErrorHandler	문법해석시에 발생한 오유 혹은 경고를 알린다.
QXmlEntityResolver	문법해석시에 외부실체들을 알리며 사용자들은 그것을 읽기기구에 넘길 대신에 외부실체들이 자체로 해결하게 한다.
QXmlDeclHandler	기타 DTD관련사건들(실례로 속성선언)을 알린다.
QXmlLexicalHandler	문서의 어휘구조(DTD의 시작, 설명문들 등)와 관련한 사건들을 알린다.

이 클라스들은 대면부를 서술하는 클라스들이다. QXmlDefaultHandler클라스는 그 모두에 대하여 《아무 일도 하지 않는》 기정실현을 제공한다. 그러므로 작성자는 사람들이 관심을 가지는 QXmlDefaultHandler함수들을 재정의할 필요만 있다.

XML입력자료를 읽는데 특수클라스 QXmlInputSource가 쓰인다.

이미 언급한것을 내놓고 표 6-5와 같은 SAX2지원클라스들은 추가적으로 유용한 기능을 제공한다.

표 6-5. SAX2지원클라스

클라스	설명
QXmlAttributes	시작요소사건의 특성(attribute)들을 넘기는데 쓰인다.
QXmlLocator	사건의 실제문법해석위치를 얻는데 쓰인다.

QXmlNamespaceSupport	읽기기구용의 이름공간지원을 실현하는데 쓰인다. 이름공간은 문법해석동작을 변경하지 않는다. 이름공간은 오직 처리함수로부터 통보된다.

2) 기능

XML읽기기구의 동작은 일정한 선택적인 기능들에 대한 지원에 의존한다. 실례로 읽기기구는 《꼬리표의 국부이름에 따라서 이름공간선언과 앞붙이들에 쓰이는 특성들을 알리는》 기능을 가질수 있다. 모든 다른 기능처럼 이것은 URI에 의해 표시되는 유일이름을 가지며 그것을 http://xml.org/sax/features/namespace-prefixes라고 부른다.

Qt SAX2실현은 읽기기구가 QXmlReader::hasFeature()함수를 사용하는 특별한 기능을 가지는가 알릴수 있다. 유효한 기능들은 QXmlReader::feature()에 의해 시험할수 있으며 QXmlReader::setFeature()에 의해 절환할수 있다.

실례를 고찰하자.

```
<document xmlns:book = 'http://trolltech.com/fnord/book/'
      xmlns    = 'http://trolltech.com/fnord/' >
```

http://xml.org/sax/features/namespace-prefixes기능을 지원하지 않는 읽기기구는 요소이름문서를 알리지만 그 특성들 즉 xmlns:book와 xmlns의 값들을 알리지 않는다. 기능 http://xml.org/sax/features/namespace-prefixes를 가지는 읽기기구는 그 기능이 설정되여있으면 이름공간속성을 알린다.

다른 기능들로서 http://xml.org/sax/features/namespace (http://xml.org/sax/features/namespace-prefixes)을 암시하는 이름공간처리와 http://xml.org/sax/features/validation (유효화오유를 통보하는 능력)을 포함한다.

SAX2은 사용자가 필요한 경우에 어떤 기능이든지 정의하고 실현하게 하며 http://xml.org/sax/features/namespace(와 http://xml.org/sax/features/namespace-prefixes)는 필수적이다. QXmlReader의 QXmlSimpleReader실현은 그것들을 지원하며 이름공간처리를 수행할수 있다.

QXmlSimpleReader는 유효화되지 않으므로 http://xml.org/sax/features/validation를 지원하지 않는다.

3) 기능을 통한 이름공간유지

앞에서 본바와 같이 이름공간처리할 때 읽기기구의 동작환경을 설정할수 있다. 이것은 http://xml.org/sax/features/namespaces와 http://xml.org/sax/features/namespace-prefixes기능들을 설정 및 해제함으로써 수행된다.

그것들은 다음과 같은 방법으로 통보조작에 영향을 준다.

① 요소와 특성들의 이름공간앞붙이와 국부부분들을 통보할수 있다.

② 요소와 특성들의 수식이름들을 통보한다.

③ QXmlContentHandler::startPrefixMapping()와 QXmlContentHandler::endPrefixMapping()이 읽기기구에 의해 호출된다.

④ 이름공간을 선언하는 특성들(즉 특성 xmlns와 xmlns:로 시작하는 특성들)을 통보한다.

다음의 요소를 고찰하자.

```
<author xmlns:fnord = 'http://trolltech.com/fnord/'
        title="Ms"
        fnord:title="Goddess"
        name="Eris Kallisti"/>
```

http://xml.org/sax/features/namespace-prefixes를 TRUE로 설정하면 읽기기구는 4가지 특성들을 통보하지만 namespace-prefixes기능을 FALSE로 설정하면 3

가지만 알려주는것과 함께 읽기기구에 《 보이지 않는 》 이름공간을 정의하는 xmlns:fnord특성을 통보한다.

http://xml.org/sax/features/namespaces란은 국부이름, 이름공간앞붙이, URI를 알린다. http://xml.org/sax/features/namespaces를 TRUE로 설정하면 문법해석기는 title을 fnord:title특성의 국부이름으로서, fnord를 이름공간앞붙이로서, http://trolltech.com/fnord/를 이름공간URI로서 알린다. http://xml.org/sax/features/namespaces가 FALSE일 때 아무것도 통보하지 않는다.

현재의 실현에서 Qt XML클라스들은 앞붙이 xmlns자체가 어떤 이름공간과 전혀 련관되지 않는다는 정의를 따른다.) 그러므로 http://xml.org/sax/features/namespaces와 http://xml.org/sax/features/namespace-prefixes를 둘다 TRUE로 설정해도 읽기기구는 xmlns:fnord에 대하여 국부이름, 이름공간앞붙이 또는 이름공간URI를 돌려주지 않는다.

이것은 xmlns를 이름공간 http://www.w3.org/2000/xmlns와 련관시키려는 W3C제안 http://www.w3.org/2000/xmlns/에 따라서 앞으로 변경될수 있다.

SAX2표준이 제안하는것처럼 QXmlSimpleReader는 기정으로 TRUE로 설정된 http://xml.org/sax/features/namespaces와 FALSE로 설정된 http://xml.org/sax/features/namespace-prefixes를 가진다.

QXmlSimpleReader::setFeature()를 리용하여 이 동작을 변경할 때 FALSE로 설정한 두 란의 결합은 옳지 않다.

2개의 특별란이 읽기기구의 출력에 어떤 영향을 주는가를 실천적으로 보기 위하여 특별란실례를 가지고 꼬리표읽기기구를 실행한다.

이상을 요약하면 QXmlSimpleReader는 표 6-6과 같은 동작을 실현한다.

표 6-6. QXmlSimpleReader의 동작

(이름공간, 이름공간앞붙이)	이름공간앞붙이와 지역부분	수식이름	앞붙이사영	xmlns 특성
(TRUE, FALSE)	Yes	Yes*	Yes	No
(TRUE, TRUE)	Yes	Yes	Yes	Yes
(FALSE, TRUE)	No*	Yes	No*	Yes
(FALSE, FALSE)	Illegal			

이 항목들의 동작은 SAX에 의하여 지정되지 않는다.

4) 속성

속성(property)은 더 일반적인 개념이다. 속성은 URI로 표시되는 유일이름을 가지지만 그 값은 void*이다. 이와 근사하게 어떤 값이 속성값으로 사용될수 있다. 이 개념은 위험성을 내포하며 형안전을 담보하는 수단도 없고 사용자는 그것들이 정확한 형을 넘긴다는것을 생각해야 한다. 속성은 읽기기구가 특별한 처리함수클라스들을 유지하는 경우에 사용할수 있다.

특별란과 속성들에 사용된 URI들은 URL(실례로 http://xml.org/sax/features/namespace)들과 같아보인다. 이것은 요구되는 자료가 이 주소에 있다는것을 의미하지 않고 순수 유일이름을 정의하는 방법이다.

누구나 읽기기구용의 새 SAX2속성들을 정의하고 사용할수 있다. 속성유지는 필수적인것이 아니다.

속성들을 설정하거나 질문하기 위하여 다음의 함수들 즉 QXmlReader::setProperty(), QXmlReader::property() 및 QXmlReader::hasProperty()이 제공된다.

3. Qt DOM클라스들

1) DOM

DOM은 XML파일의 내용과 구조를 호출하고 변경하기 위한 대면부를 제공하며 문서의 계층보기(나무보기)를 만든다. 이처럼 SAX2대면부와 대조되게 문서의 객체모형은 조작을 간단하게 만드는 문법해석후에 기억기에 상주한다.

문서나무의 모든 DOM마디들은 QDomNode의 파생클라스이다. 문서자체는 QDomDocument객체로 표시된다.

아래에 유용한 마디클라스들과 그 가능한 자식클라스들이 있다.

① QDomDocument: 가능한 자식들은 다음과 같다.
QDomElement (대체로 하나)
QDomProcessingInstruction
QDomComment
QDomDocumentType

② QDomDocumentFragment: 가능한 자식들은 다음과 같다.
QDomElement
QDomProcessingInstruction
QDomComment
QDomText
QDomCDATASection
QDomEntityReference

③ QDomDocumentType: 자식이 없다.

④ QDomEntityReference: 가능한 자식들은 다음과 같다.
QDomElement
QDomProcessingInstruction
QDomComment
QDomText
QDomCDATASection
QDomEntityReference

⑤ QDomElement: 가능한 자식들은 다음과 같다.
QDomElement
QDomText
QDomComment
QDomProcessingInstruction
QDomCDATASection
QDomEntityReference

⑥ QDomAttr: 가능한 자식들은 다음과 같다.
QDomText
QDomEntityReference

⑦ QDomProcessingInstruction: 자식이 없다.

⑧ QDomComment: 자식이 없다.

⑨ QDomText: 자식이 없다.

⑩ QDomCDATASection: 자식이 없다.

⑪ QDomEntity: 가능한 자식들은 다음과 같다.
QDomElement
QDomProcessingInstruction
QDomComment

QDomText
QDomCDATASection
QDomEntityReference
⑫ QDomNotation: 자식이 없다.

QDomNodeList와 QDomNamedNodeMap라는 두개의 집합클라스가 제공된다. QDomNodeList은 마디들의 목록이고 QDomNamedNodeMap은 순서화되지 않은 마디들의 모임이다.

QDomImplementation클라스는 DOM실현의 특성을 사용자가 질문하게 한다.

4. 이름공간

Qt XML모듈문서의 부분들은 사람들이 XML이름공간을 알고있다고 가정한다.

이름공간은 XML에 도입된 개념으로서 더욱 더 모듈적인 설계를 가능하게 한다. 이름공간의 방조에 의해 자료처리쏘프트웨어는 XML문서들에서 이름충돌을 간단히 해결할수 있다.

다음의 실례를 고찰하자.

```
<document>
<book>
  <title>Practical XML</title>
  <author title="Ms" name="Eris Kallisti"/>
  <chapter>
  <title>A Namespace Called fnord</title>
  </chapter>
</book>
</document>
```

여기서는 이름 title의 3가지 각이한 사용을 볼수 있다. 이 문서를 처리하려고 한다면 매개 title이 같은 이름을 가지고있지만 각이한 방법으로 현시하여야 하므로 문제가 생긴다.

해결대책은 book의 제목으로서 title의 첫 출현을 식별하기 위한 수단을 가지고있어야 하는것이다. 즉 book이름공간의 title요소를 사용하여 그것을 구별한다. 실례로 chapter제목이다. 례를 들면

```
<book:title>Practical XML</book:title>
```

이 경우에 book는 이름공간을 표시하는 앞붙이이다.

이름공간을 요소나 특성에 적용하기전에 그것을 선언해야 한다.

이름공간은 http://trolltech.com/fnord/book/와 같은 URI이다. 이것은 자료를 이 주소에서 사용할수 있다는것을 의미하지 않으며 URI는 단지 유일이름제공에 쓰인다.

특성과 같은 방법으로 이름공간을 선언한다. 엄격히 말해서 속성은 특성이다. 실례로 http://trolltech.com/fnord/을 문서의 기정XML이름공간 xmlns로 만들기 위하여 다음과 같이 쓴다.

```
xmlns="http://trolltech.com/fnord/"
```

http://trolltech.com/fnord/book/이름공간을 기정과 구별하기 위하여서는 앞붙이와 함께 그것을 써야 한다. 즉

```
xmlns:book="http://trolltech.com/fnord/book/"
```

이렇게 선언하는 이름공간은 적당한 앞붙이와 ":"구분기호를 앞에 련결하여 요소와 특성이름들에 적용할수 있다. book:title요소에서 이것을 이미 보았다.

앞붙이가 없는 요소이름들은 기정이름공간에 속한다. 이 규칙은 특성들에 적용하지

못한다. 앞붙이가 없는 특성은 선언된 XML이름공간에 전혀 속하지 않는다. 특성들은 늘 그것들이 나타나는 요소의 《전통적인》 이름공간에 속한다. 《전통적인》이름공간은 XML이름공간이 아니며 이것은 단순히 어떤 요소에 속하는 모든 특성이름들이 달라야 한다는것을 의미한다. 후에 특성에 XML이름공간을 할당하는 방법을 보게 된다.

앞붙이가 없는 특성들이 XML이름공간이 아니라는 사실로 인하여 특성 title (author요소에 속하는것)과, 실례로 chapter안의 title요소사이의 충돌은 없다.

실례를 통하여 이것을 명백히 하자.

```
<document xmlns:book = 'http://trolltech.com/fnord/book/'
      xmlns    = 'http://trolltech.com/fnord/' >
<book>
  <book:title>Practical XML</book:title>
  <book:author xmlns:fnord = 'http://trolltech.com/fnord/'
title="Ms"
fnord:title="Goddess"
name="Eris Kallisti"/>
  <chapter>
  <title>A Namespace Called fnord</title>
  </chapter>
</book>
</document>
```

document요소안에서는 두개의 이름공간이 선언되여있다. 기정이름공간 http://trolltech.com/fnord/는 book요소, chapter요소, 적당한 title요소와 물론 document 그 자체에 적용된다.

book:author와 book:title요소들은 URI http://trolltech.com/fnord/book/를 가지는 이름공간에 속한다.

2개의 book:author특성 title과 name에는 할당된 XML이름공간이 없다. 그것들은 오직 요소 book:author의 《전통적인》이름공간의 성원일뿐이며 이것은 실례에서 book:author안의 두개 title특성들이 금지된다는것을 의미한다.

우의 실례에서는 http://trolltech.com/fnord/이름공간으로부터 book:author:에 title특성을 추가함으로써 마지막 규칙을 우회하며 fnord:title은 book:author요소에서 선언되는 앞붙이 fnord를 가지는 이름공간으로부터 온다.

명백히 fnord이름공간은 기정이름공간으로서 같은 이름공간URI을 가진다. 그러면 왜 이미 선언한 기정이름공간을 단순히 사용하지 못하는가? 그것은 아주 복잡하기때문다.

- 앞붙이를 가지는 특성들은 XML이름공간에 전혀 속하지 않으며 지어 기정이름공간에도 속하지 않는다.
- 또한 앞붙이의 생략은 title-title충돌을 가져온다.
- 그것을 xmlns:title과 같이 쓰면 기정이름공간 xmlns를 적용할 대신에 앞붙이 title를 가지고 새 이름공간을 선언한다.

Qt XML클라스들에서 요소와 특성들은 두가지 방법으로 호출할수 있다. 즉 이름공간앞붙이와 《실제》이름(또는 국부이름)으로 이루어지는 수식이름들을 참고한다. 국부이름과 이름공간 URI를 결합하여 호출할수 있다.

XML이름공간에 대한 자세한 정보는 http://www.w3.org/TR/REC-xml-names/에서 찾아볼수 있다.

- Qt XML문서에서 사용하는 관례들

다음의 용어들은 이름공간문맥안에서 이름부분을 구별하는데 쓰인다.

- 수식이름(qualified name)은 문서에 나타나는것과 같은 이름이다. (우의 실례에

서 book:title는 수식이름이다.)

• 수식이름에서 이름공간앞붙이(namespace prefix)는 ":"의 왼쪽부분이다. (book는 book:title에서 이름공간앞붙이이다.)

• 이름의 국부부분(또는 국부이름으로 언급된다.)은 ":"의 오른쪽에 나타난다. (따라서 title은book:title의 국부부분이다.)

• namespace URI ("Uniform Resource Identifier")은 이름공간의 유일식별자이다. 이것은 URL(실례로 http://trolltech.com/fnord/)과 같아보이지만 주어진 통신규약에 따라 이름있는 주소에서 자료를 호출할수 있게 되는것을 요구하지 않는다.

":"이 없는 요소들(실례에서 chapter)은이름공간앞붙이를 가지지 않는다. 이 경우에 국부부분과 수식이름은 동등하다(즉 chapter).

5. Qt XML클라스들에서 SAX2기능의 사용실례

이 절은 XML의 이름공간과 SAX2문법해석기의 개념을 알고있는것을 전제로 한다.

여기서는 두가지 문제 즉 SAX2기능을 설정하는 방법과 Qt XML기능을 Qt GUI 응용프로그람에 통합하는 방법을 설명한다.

결과로 생기는 응용프로그람은 두개의 기능 http://xml.org/sax/features/namespace-prefixes 와 http://xml.org/sax/features/namespaces이 어떻게 설정되는가에 따라 읽기기구의 출력을 비교하게 한다. 그러기 위하여 요소들의 수식이름과 특성들, 각각의 이름공간URI들을 렬거하는 XML읽기파일의 나무보기를 표시한다.

1) 기능설정

응용프로그람의 기본프로그람으로부터 시작하자. 우선 필요한 모든 클라스들을 포함한다.

```
#include "structureparser.h"
#include <qapplication.h>
#include <qfile.h>
#include <qxml.h>
#include <qlistview.h>
#include <qgrid.h>
#include <qmainwindow.h>
#include <qlabel.h>
```

structureparser.h는 structureparser.cpp에서 실현하는 XML문법해석기의 API를 포함한다.

```
int main( int argc, char **argv )
{
  QApplication app( argc, argv );
```

보통처럼 그다음 Qt응용프로그람객체를 창조하고 지령행인수들을 거기에 넘긴다.

```
  QFile xmlFile( argc == 2 ? argv[1] : "fnord.xml" );
```

사용자가 인수로서 한개의 파일이름을 가지고 프로그람을 실행하면 파일을 처리하고 그렇지 않으면 example등록부로부터 fnord.xml 파일을 사용한다.

```
  QXmlInputSource source( &xmlFile );
```

xmlFile 을 XML입력원천으로 사용하며

```
  QXmlSimpleReader reader;
```

reader 객체의 실례를 만든다. 후에 그 기능들을 조작하여 XML자료를 읽어들이는 방법에 영향을 준다.

```
  QGrid * container = new QGrid( 3 );
```

그러면 출력현시에 대하여 고찰하자. http://xml.org/sax/features/namespace-

prefixes과 http://xml.org/sax/features/namespaces의 3가지 유효결합 즉 TRUE/TRUE, TRUE/FALSE 및 FALSE/TRUE가 있다. 관련한 출력을 나란히 표시하고 그것들을 3가지로 표식하기 위하여 3개 렬과 두 행으로 이루어지는 살창배치관리자를 만든다.

```
QListView * nameSpace = new QListView( container, "table_namespace" );
```

XML요소들을 표시하는 가장 자연스러운 방법은 나무이다. 따라서 목록보기를 사용한다. 그 이름 nameSpace는 이것이 QXmlSimpleReader의 기정환경구성인 http://xml.org/sax/features/namespaces가 TRUE, http://xml.org/sax/features/namespace-prefixes가 FALSE인 결합을 표시하는데 쓰인다는것을 가리킨다.

첫 살창항목이 nameSpace이므로 목록보기는 가상살창의 왼쪽웃구석에 나타난다.

```
StructureParser * handler = new StructureParser( nameSpace );
```

그다음 읽기기구에 의하여 읽어들인 XML자료를 취급하는 처리함수를 창조한다. 주어진 처리함수클라스 QXmlDefaultHandler가 읽기기구로부터 들어온 자료를 가지고 아무일도 하지 않을 때 그것을 정확히 사용할수 없다. 그대신에 그로부터 자체의 StructureParser를 파생시켜야 한다.

```
reader.setContentHandler( handler );
```

handler는 읽기기구용 내용처리함수로 동작한다. 편리상 오유처리함수를 등록하지 않는다. 이리하여 프로그람은 해석된 XML문서에서 닫긴 꼬리표들을 놓치는 실례에 대하여 오유를 통보하지 않는다.

```
reader.parse( source );
```

끝으로 읽기기구의 기정기능설정을 리용하여 문서를 해석한다.

```
QListView * namespacePrefix = new QListView(container, "table_namespace_prefix" );
```

이제는 같은 XML입력원천을 다른 읽기기구설정을 리용하여 해석할 준비를 한다. 출력은 둘째 QListView, namespacePrefix에서 제시된다. 이것이 container살창의 둘째 성원일 때 살창 웃행의 중간에 나타난다.

```
handler->setListView( namespacePrefix );
```

그다음 handler에 namespacePrefix목록보기안의 자료를 현시할것을 요구한다.

```
reader.setFeature( "http://xml.org/sax/features/namespace-prefixes", TRUE );
```

이제는 reader의 동작을 수정하고 http://xml.org/sax/features/namespace-prefixes를 기정의 FALSE로부터 TRUE로 변경한다. http://xml.org/sax/features/namespaces기능은 아직 기정설정값 TRUE를 가진다.

```
source.reset();
```

새로운 문법해석이 문서의 선두로부터 다시 시작하도록 만들기 위하여 입력원천을 재설정해야 한다.

```
reader.parse( source );
```

끝으로 변경된 읽기기구설정(TRUE/TRUE)을 사용하여 XML파일을 두번째로 해석한다.

```
QListView * prefix = new QListView( container, "table_prefix");
handler->setListView( prefix );
reader.setFeature( "http://xml.org/sax/features/namespaces", FALSE );
source.reset();
reader.parse( source );
```

다음으로 오른쪽우의 목록보기를 준비 및 사용하여 http://xml.org/sax/features/ namespaces가 FALSE,

http://xml.org/sax/features/namespace-prefixes가 TRUE인 기능설정값을 리용하여 읽기기구결과들을 표시한다.

```
// namespace label
(void) new QLabel( "Default:\nhttp://xml.org/sax/features/namespaces: TRUE\n"
  "http://xml.org/sax/features/namespace-prefixes: FALSE\n", container );

// namespace prefix label
(void) new QLabel( "\nhttp://xml.org/sax/features/namespaces: TRUE\n"
  "http://xml.org/sax/features/namespace-prefixes: TRUE\n", container );

// prefix label
(void) new QLabel( "\nhttp://xml.org/sax/features/namespaces: FALSE\n"
  "http://xml.org/sax/features/namespace-prefixes: TRUE\n", container );
```

container살창의 둘째 행은 우의 목록보기에 속하는 읽기기구설정을 표시할수 있는 3개의 표식자들로 채워진다.

```
app.setMainWidget( container );
container->show();
return app.exec();
}
```

Qt GUI프로그람에서와 같은 살창은 응용프로그람의 기본창문부품으로서 동작하고 표시된다. 그 후에 GUI의 사건순환고리에 들어간다.

2) 처리함수API

처리함수클라스 StructureParser의 API를 간단히 고찰하자.

```
#include <qxml.h>
#include <qptrstack.h>

class QListView;
class QListViewItem;
class QString;
class StructureParser: public QXmlDefaultHandler
{
```

아무것도 하지 않는 처리함수를 실현하는 QXmlDefaultHandler클라스로부터 그것을 파생시킨다.

```
public:
 StructureParser( QListView * );
 bool startElement( const QString&, const QString&, const QString& ,
      const QXmlAttributes& );
 bool endElement( const QString&, const QString&, const QString& );
```

이것은 사실상 요구되는 기능만 간단히 실현한다. 실례에서는 인수로서 QListView를 가지는 구성자, 요소시작꼬리표들이 출현할 때 실행할 함수(QXmlContentHandler로부터 계승), 그리고 완료꼬리표가 발생시키는 함수이다.

지금 실현해야 하는것은 내용조종이다.

```
void setListView( QListView * );
```
또한 출력의 목록보기를 선택하는 함수를 가진다.
```
private:
QPtrStack<QListViewItem> stack;
```
기억기에 모든 요소와 특성을 보관하기 위한 객체모형이 없는 SAX2문법해석기를 작성한다. 그렇지만 요소와 특성들을 나무구조로 현시하기 위하여 아직은 닫겨지지 않은 모든 요소의 궤적을 보관하여야 한다.

그러기 위하여 QListItem들의 LIFO탄창을 사용한다. 하나의 요소는 시작끄리표가 나타나고 탄창에 추가되고 완료타그가 해석되면 곧 삭제된다.
```
QListView * table;
};
```
이와는 별도로 현재 사용한 목록보기를 포함하는 성원변수를 정의한다.

3) 처리함수자체

이제는 API를 정의하였으므로 관련한 함수들을 실현해야 한다.
```
#include "structureparser.h"

#include <qstring.h>
#include <qlistview.h>
StructureParser::StructureParser( QListView * t )
    : QXmlDefaultHandler()
{
```
우선 인수로서 목록보기지적자를 가지는 구성자가 있다.
```
setListView( t );
}
```
여기서 수행해야 할 일은 사용하기전에 인수 QListView를 준비하는것이다. 이것을 setListView()함수를 가지고 수행한다.
```
void StructureParser::setListView( QListView * t )
{
table = t;
```
우선 인수를 보관한다.
```
table->setSorting( -1 );
```
문서에 나타나는것처럼 요소들을 가령 자모순으로 분류하지 않고 렬거한다. 그것은 분류기능을 완전히 차단하기 위해서다.
```
table->addColumn( "Qualified name" );
table->addColumn( "Namespace" );
}
```
현재 목록보기는 두개의 렬로 이루어진다. 즉 하나는 요소 혹은 특성의 수식이름, 다른 하나는 그 이름공간URI. 렬들은 왼쪽에서 오른쪽으로 추가되고 인수로서 제목을 가진다.

이제는 XML내용조종을 취급한다.
```
bool StructureParser::startElement( const QString& namespaceURI,
const QString& ,const QString& qName, const QXmlAttributes& attributes)
{
```
우연히 요소의 시작끄리표와 만날 때 처리함수는 실제 작업을 수행한다. startElement 가 4개의 인수를 가지고 호출되여도 오직 3개의 궤적 즉 요소의 이름공간URI, 수식이름 및 특성만 보유한다. 요소에 할당된 이름공간이 없거나 읽기기구의

기능설정이 처리함수에 이름공간URI들을 전혀 제공하지 않는다면 namespaceURI 는 빈 문자렬을 포함한다.

변수를 둘째 인수에 대입하지 않으며 요소의 국부이름에 흥미를 가지지 않는다.

```
QListViewItem * element;
```

요소가 발생할 때마다 목록보기에 표시하기 위하여 QListViewItem변수를 정의한다.

```
if ( ! stack.isEmpty() ){
 QListViewItem *lastChild = stack.top()->firstChild();
```

요소stack 이 비여있지 않는한 현재 요소는 탄창의 제일 웃끝 요소의 자식이다. 이처럼 첫 렬에 새 요소의 수식이름, 둘째 렬에 그에 따르는 이름공간URI를 가지는 QPtrStack::stack.top()의 자식으로 새로운 QListViewItem을 창조한다.

보통 QListViewItem은 첫 자식으로 삽입된다. 이것은 요소들을 반대순서로 얻는다는것을 의미한다. 그러므로 우선 QPtrStack::stack.top()요소의 마지막 자식을 탐색하여 요소뒤에 삽입한다.

유효XML문서에서 이것은 문서의 뿌리를 제외한 모든 요소에 적용한다.

```
 if ( lastChild ) {
  while ( lastChild->nextSibling() )
   lastChild = lastChild->nextSibling();
 }
 element = new QListViewItem( stack.top(), lastChild, qName, namespaceURI );
} else {
 element = new QListViewItem( table, qName, namespaceURI );
}
```

뿌리요소는 QListViewItem탄창에 넘겨야 할 첫 요소이므로 따로 조종해야 한다. 그 목록보기항목은 table목록보기자체의 자식이다.

```
stack.push( element );
```

이제는 탄창꼭대기에 요소의 목록보기항목을 넣는다.

```
element->setOpen( TRUE );
```

기정으로 QListView는 닫긴 마디들을 모두 표시한다. 그다음 사용자는 +그림기호를 찰칵하여 자식항목들을 볼수 있다.

그러나 프로그람을 실행할 때 전체요소나무를 한번에 보려고 하므로 매개의 목록보기항목을 수동적으로 연다.

```
if ( attributes.length() > 0 ) {
```

요소가 특성을 가진다면 어떻게 하겠는가?

```
 for ( int i = 0 ; i < attributes.length(); i++ ) {
  new QListViewItem( element, attributes.qName(i), attributes.uri(i) );
 }
}
```

그 매개에 대하여 새로운 목록보기항목을 창조하여 특성의 수식이름과 관련이름공간URI들을 표시한다. 명백히 attribute는 현재 element의 자식이다.

```
 return TRUE;
}
```

읽기기구에서 오유의 발생을 방지하기 위하여 요소의 시작꼬리표를 성공적으로 취급할 때 TRUE를 돌려주어야 한다.

```
bool StructureParser::endElement( const QString&, const QString&, const QString& )
```

```
{
 stack.pop();
```
요소의 닫긴 꼬리표를 만날 때마다 더는 자식이 없으면 탄창에서 목록보기항목을 삭제해야 한다.
```
 return TRUE;
}
```
이로서 완료한다.

6. Qt SAX2클라스들의 사용례- 아주 작은 문법해석기

여기서는 지령행에 XML문서의 모든 요소의 이름을 출력하는 작은 실례읽기기구를 보여준다. 요소이름은 그 겹쌓임준위에 대응한다.

우리가 관심을 가지는 처리함수클라스들의 3개 함수 즉 QXmlContentHandler::startDocument(), QXmlContentHandler::startElement() 그리고 QXmlContentHandler::endElement()를 실현해야 한다.

이를 위하여 QXmlDefaultHandler의 파생클라스를 사용한다. (특수한 처리함수클라스들은 모두 추상적이고 기정처리함수클라스는 문법해석동작을 변경하지 않는 실현을 제공한다.)

```
#ifndef STRUCTUREPARSER_H
#define STRUCTUREPARSER_H

#include <qxml.h>

class QString;

class StructureParser : public QXmlDefaultHandler
{
public:
 bool startDocument();
 bool startElement( const QString&, const QString&, const QString& , const QXmlAttributes& );
 bool endElement( const QString&, const QString&, const QString& );

private:
 QString indent;
};
#endif
```
정확한 들여쓰기를 얻는데 사용하는 비공개보조변수 indent를 제외하면 새로운 StructureParser클라스에서 특별한것은 없다.

실현은 간단하다.
```
#include "structureparser.h"
#include <stdio.h>
#include <qstring.h>
```
우선 QXmlContentHandler::startDocument()를 재정의한다.
```
bool StructureParser::startDocument()
{
 indent = "";
```

```
return TRUE;
}
```

문서의 선두에서는 들여쓰기없이 뿌리요소를 출력하기 위하여 단순히 indent를 빈 문자렬로 설정한다. 또한 문법해석기가 오유를 통보하지 않고 계속하도록 TRUE를 돌려준다.

문법해석기가 요소의 시작꼬리표를 만나서 그것을 출력할 때 통보하기 위하여 QXmlContentHandler::startElement()를 재정의해야 한다.

```
bool StructureParser::startElement( const QString&, const QString&, const QString& qName, const QXmlAttributes& )
{
printf( "%s%s\n", (const char*)indent, (const char*)qName );
indent += "    ";
return TRUE;
}
```

앞에 들여쓰기가 있는 이름뒤에 행중지가 출력된다. 엄격히 말해서 qName은 이름공간을 표시하는 마지막 앞붙이가 없는 국부요소이름을 포함한다.

또 하나의 요소가 현재요소의 완료꼬리표앞에 있다면 그것은 들여써야 한다. 그러므로 indent문자렬에 4개의 공백을 추가한다.

끝으로 문법해석기가 오유없이 계속하도록 하기 위하여 TRUE를 돌려준다.

마지막으로 추가해야 할 기능은 완료꼬리표가 발생할 때 문법해석기의 동작이다. 이것은 QXmlContentHandler::endElement()의 재정의를 의미한다.

```
bool StructureParser::endElement( const QString&, const QString&, const QString& )
{
indent.remove( (uint)0, 4 );
return TRUE;
}
```

그때 명백히 startElement()에 추가한 4개의 공백만큼 indent문자렬이 줄어들어야 한다.

문법해석기를 실현하였으므로 main()프로그람을 쓸수 있다.

```
#include "structureparser.h"
#include <qfile.h>
#include <qxml.h>
#include <qwindowdefs.h>

int main( int argc, char **argv )
{
 if ( argc < 2 ) {
  fprintf( stderr, "Usage: %s <xmlfile> [<xmlfile> ...]\n", argv[0] );
  return 1;
 }
```

이 검사는 지령행으로부터 시험하려는 파일들의 렬을 얻을수 있다는것을 담보한다.

```
StructureParser handler;
```

다음 걸음은 StructureParser의 실례를 창조하는것이다.

```
QXmlSimpleReader reader;
reader.setContentHandler( &handler );
```

그다음 읽기기구를 설정한다. StructureParser클라스가 QXmlContentHandler 기능을 취급할 때만 그것을 우리가 선택한 내용처리함수로서 등록한다.

```
  for ( int i=1; i < argc; i++ ) {
```

지령행인수로서 주어진 모든 파일들을 성과적으로 취급한다.

```
    QFile xmlFile( argv[i] );
    QXmlInputSource source( &xmlFile );
```

그다음 문법해석하려는 XML파일용의 QXmlInputSource를 창조한다.

```
    reader.parse( source );
```

이제는 입력원천을 가지고 문법해석을 시작한다.

```
  }
  return 0;
 }
```

다음의 XML파일에 대하여 프로그람을 실행하면

```
<animals>
<mammals>
  <monkeys> <gorilla/> <orangutan/> </monkeys>
</mammals>
<birds> <pigeon/> <penguin/> </birds>
</animals>
```

다음의 출력을 생성한다.

```
animals
 mammals
  monkeys
   gorilla
   orang-utan
 birds
  pigeon
  penguin
```

그러나 실례로 자기의 XML시험파일안에서 <와 요소이름사이에 공백을 삽입하면 정확한 결과를 생성하지 않는다. 이것을 방지하기 위하여 늘 QXmlReader::setErrorHandler()를 사용하는 오유처리함수를 설치해야 한다. 이것은 사용자에게 문법해석오유들을 통보한다.

제7장. 국제화

제1절. Qt에 의한 국제화

응용프로그람의 국제화는 여러 나라 사람들이 응용프로그람을 사용할수 있게 하는 과정이다.

일부 경우에 국제화는 쉽다. 실례로 오스트랄리아 혹은 영국사용자가 호출할수 있는 미국응용프로그람제작은 얼마간의 철자수정만을 하면 된다. 그러나 일본사용자들이 사용가능한 미국응용프로그람, 도이췰란드사용자들이 사용가능한 조선응용프로그람작성은 쏘프트웨어가 여러가지 언어에서 조작할뿐아니라 여러가지 입력수법, 문자부호화와 표시관례를 사용할것을 요구한다.

Qt는 개발자들이 불편없이 국제화를 가능하게 하려고 한다. Qt의 모든 입력창문부품들과 본문그리기메쏘드들은 모든 지원언어들을 위한 내부(built-in)기능을 제공한다. 내부서체엔진은 동시에 여러가지 문서체계의 문자들을 담은 본문을 정확히, 훌륭히 표현하는 능력이 있다.

Qt는 현재 쓰이고있는 많은 언어들을 유지한다. 특히

- 모든 동아시아언어(중국어,일어와 조선어)
- 모든 서유럽언어(라틴어사용)
- 아랍어
- 시릴어(로어)
- 그리스어
- 고대헤브라이어
- 타이와 라오스어
- 특별한 처리를 요구하지 않는 유니코드 3.2에서 모든 스크립트

Windows NT/2000/XP와 Xft(의뢰자측 서체지원)를 갖춘 Unix/X11에서는 다음의 언어들도 지원한다.

벵갈어, Devanagari, Dhivehi (Thaana), Gujarati, Gurmukhi, Kannada, 크레르어,

말라이어(X11에서만), Myanmar (X11에서만), 수리아어, 타밀어, Telugu, 티베트어(X11에서만)

다음 문서체계들의 대부분은 구체적인 특성들을 제시한다.

- **특별한 행중지조작.** 일부 아시아언어들은 단어들사이에 공백이 없이 쓴다. 행중지는 중어, 일본어와 조선어에서처럼(례외를 가지고) 매 문자뒤에 있을수 있고 혹은 타이어처럼 단어경계후에 있을수 있다.
- **쌍방향 쓰기.** 아랍어와 고대 헤브라이어는 수자와 왼쪽에서 오른쪽으로 쓰는 수자들과 영어를 제외하고 오른쪽에서 왼쪽으로 쓴다.
- **공백이나 구별표식없음**(유럽언어들에서 악쎈트 혹은 모음변화). 일부 언어들은 이 표식들을 광범히 사용하며 일부 문자들은 발음을 명백하게 하기 위해 한번에 한개이상의 표식을 할수 있다.
- **묶기.** 특수한 상황에서 일부 문자쌍들은 묶기(ligature)를 형성하는 결합된 글리프로 교체되였다. 일반적인 실례는 미국식자판과 유럽문헌들에서 사용되는 fl와 fi묶기(ligature)이다.

Qt는 우에서 렬거한 모든 구체적인 특성들에 주의를 돌린다. 보통 Qt의 입력창문부품들(실례로 QLineEdit, QTextEdit과 파생클라스들)과 Qt표시창문부품들(례들어 QLabel)을 사용하는한 이 특성들에 대하여 걱정하지 않아도 된다.

이 문서체계에 대한 지원은 프로그람작성자들에게 투명하며 완전히 Qt본문엔진안에 은페된다. 이것은 다음의 몇가지 경우를 제외하고 특별한 언어에서 사용되는 문서체계에 대한 지식이 없어도 된다는것을 의미한다.

- QPainter::drawText(int x, int y, const QString &str)는 항상 x,y파라메터에 의해 지정되는 위치에 왼쪽끝을 맞춘 문자렬을 그린다. 이것은 항상 왼쪽으로 정렬된 문자를 준다. 보통 아랍어와 헤브라이어 응용프로그람문자렬들은 오른쪽끝에 맞추므로 이러한 언어들에서는 언어에 따라 맞추어쓰는 QRect를 가지는 drawText()를 사용한다.
- 자체의 본문입력조종을 쓸 때 QFontMetrics::charWidth()를 사용하여 문자렬에서 한 문자의 폭을 결정한다. 일부 언어들(아랍어 혹은 인디아지역의 언어들)에서 글리프의 너비와 형태는 주위문자들에 따라 변한다. 입력조종을 쓸 때 보통 사용하려고 하는 스크립트에 대한 일정한 지식을 요구한다. 일반적으로 가장 쉬운 방법은 QLineEdit 혹은 QTextEdit의 파생클라스를 만드는것이다.

다음은 Qt에서 지원하는 국제화(i18n)의 상태들에 대한 정보를 준다.

또한 《Qt프로그람개발도구》2장을 참고할수 있다.

1. 절차

Qt에 의한 다중가동환경 국제화쏘프트웨어작성은 유연하고 증분적인 과정이다. 쏘프트웨어는 다음의 단계들에서 국제화될수 있다.

1) 모든 사용자보임본문에서 QString사용

QString은 내부적으로 유니코드부호화를 사용하기때문에 매개 언어는 비슷한 본문처리조작에 의하여 명백히 처리될수 있다. 또한 사용자들에게 본문을 제시하는 모든 Qt함수들은 파라메터로서 QString을 가지기때문에 char*를 QString으로 변환하기 위한 추가적인 작업이 없다.

《 프로그람작성자공간 》(QObject이름과 파일형식화본문 등)에 있는 문자들은 QString을 사용할 필요가 없으며 전통적으로 쓰이는 char* 혹은 QCString클라스면 충분하다.

유니코드를 사용한다고 알리는것이 믿음직하지 않으면 QString과 QChar가 전통적인 C의 조잡한 const char*와 char의 간단한 판들이 좋다.

2) 모든 리터럴본문에서 tr()사용

자기의 프로그람이 사용자들에게 현시할 본문으로써《인용된 본문》을 사용할 때마다 QApplication::translate()함수에 의해 처리하는것이 안전하다. 이것을 달성하는데 필수적으로 요구되는것은 QObject::tr()를 사용하는것이다. 례들어 LoginWidget가 QWidget의 파생클라스라고 가정하자.

```
LoginWidget::LoginWidget()
{
  QLabel *label = new QLabel( tr("Password:"), this );
  ...
}
```

만일 인용된 본문이 QObject파생클라스의 성원함수가 아니면 적당한 클라스의 tr()함수나 QApplication::translate()함수를 직접 사용한다.

```
void some_global_function( LoginWidget *logwid )
{
  QLabel *label = new QLabel( LoginWidget::tr("Password:"), logwid );
}
```

```
void same_global_function( LoginWidget *logwid )
{
  QLabel *label = new QLabel(qApp->translate("LoginWidget", "Password:"), logwid );
}
```

만일 함수밖에서 본문을 완전히 번역하여야 한다면 그것을 방조하는 2개의 마크로가 있다. 그것들은 단지 아래에 서술한 *lupdate*편의프로그람에 의해 추출본문을 표식한다. 마크로들을 본문으로 (문맥없이) 전개한다.

QT_TR_NOOP()의 실례:

```
QString FriendlyConversation::greeting( int greet_type )
{
  static const char* greeting_strings[] = {
    QT_TR_NOOP( "Hello" ),
    QT_TR_NOOP( "Goodbye" )
  };
  return tr( greeting_strings[greet_type] );
}
```

QT_TRANSLATE_NOOP()의 실례:

```
static const char* greeting_strings[] = {
  QT_TRANSLATE_NOOP( "FriendlyConversation", "Hello" ),
  QT_TRANSLATE_NOOP( "FriendlyConversation", "Goodbye" )
};

QString FriendlyConversation::greeting( int greet_type )
{
  return tr( greeting_strings[greet_type] );
}

QString global_greeting( int greet_type )
{
  return qApp->translate( "FriendlyConversation", greeting_strings[greet_type] );
}
```

마크로 QT_NO_CAST_ASCII정의를 가진 쏘프트웨어를 콤파일하여 const char*를 QString으로 자동변환할수 없으면 자기가 놓친 문자렬을 찾는것이 아주 좋다(QString::fromLatin1()참고.) 변환을 허용하지 않으면 프로그람작성이 좀 시끄러워진다.

자기의 원천언어가 Latin-1밖의 문자를 사용하면 QObject::tr()보다 QObject::trUtf8()이 더 편리하다는것을 알게 된다. tr()는 QApplication::defaultCodec()에 의존되므로 QObject::trUtf8()보다 더 불리하다.

3) 지름값들에 QKeySequence()사용

Ctrl+Q 혹은 Alt+F와 같은 지름건값(accelerator value)들도 번역해야 할 필요가 있다. 응용프로그람에 "Quit"에 해당한 CTRL+Key_Q를 고정코드로 작성하면 번역기는 그것을 무시할수 없다. 정확한 표현은 다음과 같다.

```
QPopupMenu *file = new QPopupMenu( this );
```

```
file->insertItem( tr("&Quit"), this,
         SLOT(quit()),QKeySequence(tr("Ctrl+Q", "File|Quit")) );
```
4) 동적본문에 QString::arg ()사용

QString ::arg()함수는 인수를 대신하는 간단한 수단을 제공한다.

```
void FileCopier::showProgress( int done, int total, const QString& current_file )
{
  label.setText( tr("%1 of %2 files
      copied.\nCopying: %3").arg(done).arg(total).arg(current_file) );
}
```
일부 언어들에서 인수의 순서를 바꾸어야 할 때가 있는데 이것은 %인수들의 순서를 바꿈으로써 쉽게 해결될수 있다. 례들어 다음

```
QString s1 = "%1 of %2 files copied. Copying: %3";
QString s2 = "Kopierer nu %3. Av totalt %2 filer er %1 kopiert.";
qDebug( s1.arg(5).arg(10).arg("somefile.txt").ascii() );
qDebug( s2.arg(5).arg(10).arg("somefile.txt").ascii() );
```
은 영어와 노르웨이어로 정확한 출력문

5 of 10 files copied. Copying: somefile.txt

Kopierer nu somefile.txt. Av totalt 10 filer er 5 kopiert.

을 생성한다.

5) 번역문생성

응용프로그람을 통해 tr()를 사용하면 프로그람에서 사용자보임본문의 번역생성을 시작할수 있다.

《 Qt도구의 사용법 》2장에서는 Qt의 번역도구들인 *Qt Linguist*과 *lupdate*, *lrelease*에 대한 자세한 정보를 제공한다.

Qt응용프로그람의 번역은 3단계로 이루어진다.

① *lupdate*를 실행하여 Qt응용프로그람의 C++원천코드로부터 번역해야 할 본문을 끌어내고 번역기용통보파일(.ts파일)에 결과를 보관한다. 이 편의프로그람은 tr()구조와 우에서 서술한 QT_*_NOOP마크로들을 인식하여 .ts파일들(보통 언어당 1개씩)을 생성한다.

② *QT Linguist*를 사용하여 .ts파일에서 원천본문들을 위한 번역문을 제공한다. .ts파일들이 XML형식으로 되여있으므로 자체로 편집할수도 있다.

③ *lrelease*를 실행하여 .ts파일로부터 사용하는데 적합한 가벼운 통보파일(.qm파일)을 얻는다. .ts파일들을 《원천파일》로서, .qm파일들을 《목적파일》들로 고찰한다. 번역기는 .ts파일들을 편집하지만 응용프로그람의 사용자들은 .qm파일들만 요구한다. 두 종류의 파일들은 가동환경과 지역에 의존하지 않는다.

일반적으로 응용프로그람을 개정할 때마다 이 단계들을 반복한다. lupdate편의프로그람은 앞선 출하물로부터 번역을 재사용하는데 제일 좋다.

```
HEADERS      = funnydialog.h \
 wackywidget.h
SOURCES      = funnydialog.cpp \
 main.cpp \
 wackywidget.cpp
FORMS        = fancybox.ui
TRANSLATIONS  = superapp_dk.ts \
 superapp_fi.ts \
```

```
    superapp_no.ts \
    superapp_se.ts
```

lupdate나 lrelease를 실행할 때 지령행인수로서 프로젝트파일의 이름을 주어야 한다.

이 실례에서는 4개 언어 즉 단마르크어, 핀란드어, 노르웨이어, 스웨리예어들이 지원된다. qmake를 사용하면 보통 lupdate를 위한 특정한 프로젝트파일이 필요없고 qmake프로젝트파일에 TRANSLATIONS항목만 추가하면 잘 동작한다.

응용프로그람에서는 사용자언어에 알맞는 번역파일들을 QTranslator::load()하고 QApplication::installTranslator()를 리용하여 설치해야 한다.

낡은 Qt도구들(findtr, msg2qm, mergetr)을 사용하여왔다면 qm2t를 사용하여 낡은 .qm파일들을 변환할수 있다.

linguist, lupdate 그리고 lrelease는 Qt가 설치된 기초등록부의 bin보조등록부에 설치된다. Qt Linguist에서 Help|Manual을 선택하여 사용자지도서를 호출하면 여기에는 그 시작을 위한 지도서가 있다.

이 편의프로그람들은 .qm파일들을 창조하는 편리한 방법을 제공하는 한편 .qm파일들을 쓰는 체계면 충분하다. QTranslator::insert()에 의하여 QTranslator에 번역문을 추가하는 응용프로그람을 만든 다음 QTranslator::save()에 의하여 .qm파일들을 쓴다. 이 방법으로 번역문을 자기가 선택한 원천으로부터 꺼낼수 있다.

Qt자체는 목표언어들로 번역하는데 필요되는 400개이상의 문자렬을 포함한다. 다른 언어로 번역하기 위한 형판뿐아니라 $QTDIR/translations에서 프랑스어와 도이칠란드어용 번역파일을 볼수 있다. (이 등록부는 또한 유지되지 않지만 유용한 번역문을 추가적으로 포함한다.)

전형적으로 응용프로그람의 main()함수는 다음과 같다.

```
int main( int argc, char **argv )
{
  QApplication app( argc, argv );

  // Qt용 번역파일
  QTranslator qt( 0 );
  qt.load( QString( "qt_" ) + QTextCodec::locale(), "." );
  app.installTranslator( &qt );

  // 응용프로그람문자렬용 번역파일
  QTranslator myapp( 0 );
  myapp.load( QString( "myapp_" ) + QTextCodec::locale(), "." );
  app.installTranslator( &myapp );

  ...

  return app.exec();
}
```

6) 부호화지원

QTextCodec클라스와 QTextStream을 사용하는 편의프로그람들은 사용자들의 자료에 대한 수많은 입력과 출력부호화를 쉽게 지원한다. 응용프로그람이 기동할 때 콤퓨터가 있는 지역은 서체선택에서 본문표시 8bit본문입출력과 문자입력 등은 8bit자료를 처리할 때 사용되는 8bit부호화를 결정하게 된다.

응용프로그람은 때때로 기정의 국부적인 8bit부호화가 아닌 다른 부호화를 요구할

수 있다. 례들어 Cyrillic KOI8-R지역(사실상 로씨야의 표준지역)에서 응용프로그람은 ISO8859-5부호화의 씨릴어로 출력하여야 한다.

```
QString string = ...; // 유니코드본문
QTextCodec* codec = QTextCodec::codecForName( "ISO 8859-5" );
QCString encoded_string = codec->fromUnicode( string );

...; // use encoded_string in 8bit operations
```

유니코드를 국부적인 8bit부호화로 변환하는데 지름건이 쓸모있고 QString의 local8Bit()메쏘드는 그러한 8bit자료를 돌려준다. 다른 유용한 지름건은 utf8()메쏘드인데 이것은 8bit UTF-8부호화로 된 본문을 돌려준다. 이것은 유니코드정보를 전적으로 유지하는 한편 본문이 완전히 US-ASCII라면 일반US-ASCII처럼 보이게 한다.

다른 변환방법으로서 QString::fromUtf8()와 QString::fromLocal8Bit()편의함수들 혹은 일반코드가 있는데 그 코드를 ISO 8859-5 Cyrillic로부터 유니코드에로의 변환에서 보여준다:

```
QCString encoded_string = ...; // Some ISO 8859-5 encoded text.

QTextCodec* codec = QTextCodec::codecForName("ISO 8859-5");
QString string = codec->toUnicode(encoded_string);

...; // Use string in all of Qt's QString operations.
```

리상적으로 유니코드입출력장치는 세계의 여러 사용자들사이에 문서의 이식성을 최대화하기 위해 사용되지만 실제상 이것은 사용자들이 현존문서들을 처리할 필요가 있는 적당한 부호화를 모두 지원하는것이 유리하다. 일반적으로 유니코드(UTF-16혹은 UTF-8)는 임의의 사람들사이에 정보를 전송하는데 가장 유리하며 언어 혹은 민족집단안에서는 국부표준화가 더 적합하다. 지원하여야 할 가장 중요한 부호화는 QTextCodec::codecForLocale()가 돌려주는것인데 이것은 사용자가 다른 사람들이나 응용프로그람들과 통신하는데 필요한것이다. (이것은 local8Bit()에 의해 사용되는 codec이다.)

Qt는 더 빈번히 사용되는 부호화의 대부분을 원만히 지원한다(QTextCodec 참고).

일부 경우와 적게 사용되는 부호화를 위해 자체의 QTextCodec파생클라스를 쓰는것이 필요하다. 긴급성에 따라서 Trolltech기술적지원을 요구하거나 qt-interest우편목록에 부탁하여 그 부호화지원에 대하여 이미 동작하고있는가 확인한다. 림시대책은 QTextCodec::loadCharmapFile()함수를 사용하여 자료구동형codec를 구축하는것이다. 그러나 이 수법은 특히 동적으로 적재된 서고들에서 기억기와 속도에서 결함을 가진다. (자체의 QTextCodec를 쓰는 방법은 QTextCodec클라스를 참고하시오.)

7) 국부화

국부화는 지역적편리를 도모하는 처리로서 례들어 지역적으로 제기되는 형식화를 리용한 날자와 시간표시를 들수 있다. 그러한 국부화는 적당한 tr()문자렬을 사용하여 달성할수 있다.

```
void Clock::setTime(const QTime& t)
{
  if ( tr("AMPM") == "AMPM" ) {
    // 12시간 시계
  } else {
    // 24시간 시계
  }
```

```
  }
```

실례에서 영어를 사용하는 나라들에서 12시간단위의 시계를 사용하는것으로 보고 《AMPM》이라고 번역하였지만 유럽에서는 그것을 다르게(어떤곳에서는 《EU》) 번역하며 이것은 24시간단위의 시계를 코드에서 사용하게 한다.

화상의 국부화는 권고하지 않는다. 지역사투리나 그에 맞는 은유에 기초하는것보다 모든 지역들에 한해서 적당한 지우기그림기호들을 선택한다.

2. 동적번역

QT Linguist와 같은 일부 응용프로그람들에서는 실행전기간 사용자언어선택의 변경을 지원할수 있어야 한다. 체계언어변화를 알아야 할 매개 창문부품에는 languageChange()라고 부르는 공개처리부를 실현한다. 이 처리부에서는 보통 방법으로 QObject::tr(){tr()}를 사용하여 창문부품이 현시할 본문을 갱신해야 한다. 례들어

```
void MyWidget::languageChange()
{
  titleLabel->setText(tr("Document Title"));
  ...
  okPushButton->setText(tr("&OK"));
}
```

QWidget파생클라스에 해당한 기정사건처리함수는 LanguageChange사건에 응답하며 필요할 때 이 처리부를 호출한다. 다른 응용프로그람부분품들은 창문부품들이 자체로 갱신하도록 이 처리부에 신호들을 련결할수 있다.

3. 체계지원

Qt만이 동작하는 일부 조작체계와 창문체계들은 유니코드에 대해 제한된 기능을 가진다. 기초하고있는 체계에서 유효한 지원의 수준은 일반적으로 Qt응용프로그람들이 가동환경에 고유한 제한과 관련되지 않아도 Qt가 이 가동환경들에 제공할수 있는 지원에서 일정한 영향을 준다.

1) Unix/X11

- 지역지향서체와 입력방식들. Qt는 이것들을 숨기고 유니코드입출력방식을 제공한다.
- UTF-8과 같은 파일체계관례는 일부 Unix변종들에서 개발중에 있다. 모든 Qt파일함수들은 유니코드를 허용하지만 국부적 8bit부호화로 파일이름을 변환하며 이것은 Unix관례이다. (다른 부호화를 알려면 QFile::setEncodingFunction()을 참고하시오.)
- 파일 I/O는 기정으로 QTextStream에서 유니코드선택을 가지는 8bit부호화이다.

2) Windows

- Qt는 입력방법, 서체, 오려둠판, 끌어다놓기와 파일이름을 비롯하여 충분한 유니코드지원을 제공한다.
- 파일I/O는 기정으로 QTextStream에서 유니코드선택을 가지는 Latin-1이다. 일부 Windows프로그람들은 높은 수준의 규약이 없는것으로 하여 유니코드표준에 의해 서술되여있더라도 비그엔디안 유니코드본문파일들을 리해하지 못한다.
- MFC 혹은 일반winlib를 리용하여 작성한 프로그람들과 달리 Qt프로그람들은 Windows 95/98과 Windows NT사이에서 이식할수 있다. 유니코드를 지원하기 위한 다른 2진파일들을 요구하지 않는다.

4. X11에서 지역에 대한 알아두기

수많은 Unix배포물은 일부 지역에 대한 부분적인 지원만 포함한다. 실례로 /usr/share/locale/ja_JP.EUC등록부를 가지고있으면 이것은 일본어본문을 현시할수 있다는것을 의미하지 않으며 JIS부호화된 서체들 (혹은 유니코드서체들)도 요구하며

/usr/share/locale/ja_JP.EUC등록부는 완전성을 요구한다. 가장 좋은 결과들에 대하여서는 자기의 체계제작자로부터 받은 완전한 지역들을 사용한다.

5. 관련한 Qt클라스들

다음의 클라스들은 Qt응용프로그람들의 국제화와 관련되여있다.

표 7-1. 국제화관련클라스들

QBig5Codec	Big5부호화에 대한 변환
QEucJpCodec	EUC-JP문자모임변환
QEucKrCodec	EUC-KR문자모임변환
QGb18030Codec	중어GB18030/GBK/GB2312부호화에 대한 변환
QGb2312Codec	중어Chinese GB2312부호화에 대한 변환
QGbkCodec	중어Chinese GBK부호화에 대한 변환
QHebrewCodec	시각적으로 순서화된 Hebrew에 대한 변환
QJisCodec	JIS문자모임에 대한 변환
QSjisCodec	Shift-JIS에 대한 변환
QTextCodec	본문부호화사이의 변환
QTextDecoder	상태해신기
QTextEncoder	상태부호기
QTranslator	본문출력을 위한 국제화지원
QTranslatorMessage	번역기통보문과 그 속성들
QTsciiCodec	타밀TSCII부호화에 대한 변환

제2절. 유니코드

유니코드는 여러바이트문자모임으로서 모든 주요 콤퓨터가동환경에서 이식가능하고 세계의 대부분의 나라들에서 상당한 신용을 가지고있다. 또한 그것은 하나의 지역으로 표시되며코드페지나 그밖의 복잡성을 가지지 않으므로 쏘프트웨어를 쓰고 시험하기 쉽게 한다. 여러 가동환경에서 유니코드와 경쟁하는 문자모임은 없다. 이러한 리유로 Trolltech는 유니코드를 Qt(2.0판부터)용의 원시적인 문자모임으로서 사용한다.

1. 웨브상에서 유니코드에 대한 정보

유니코드협회(Consortium)는 사용가능한 수많은 문서를 가지고있다. 여기에는 다음과 같은것들도 있다.

- 유니코드에 대한 기술적소개
- 표준용 홈페지

2. 표준

표준의 현재 판은 3.2이다

- 유니코드표준, 3.2.판.
- 유니코드표준, 2.0.판.

3. Qt에서 유니코드

Qt와 Qt를 사용하는 대부분의 응용프로그람들에서 대부분 혹은 모든 사용자보임문자렬은 유니코드로 보관된다. Qt는 다음과 같은것을 제공한다.

- 파일입출력을 위한 유산부호화에 대한 호상번역(QTextCodec와 QtextStream 참고).
- 입력방식과 8bit건반입력으로부터의 번역.

• 화면상의 현시를 위하여 유산문자모임들로의 번역.

• 문자렬클라스 QString은 유니코드문자들을 보관하며 US-ASCII에 대한 고속의 (캐쉬된) 변환과 모든 보통문자렬조작을 비롯한 C문자렬들로부터의 이식을 지원한다.

• 적당한 유니코드인식창문부품들.

• Windows에서 유니코드지원탐지, 그러므로 Qt는 본래유니코드를 유지하지 않고 있는Windows가동환경에서도 유니코드를 제공한다.

유니코드로부터 완전히 덕을 보기 위하여 사용자가 볼수 있는 모든 문자렬에 QString을 사용하고 QTextStream을 사용하여 본문파일입출력을 모두 수행할것을 권고한다. 자기가 작성하는 사용자정의창문부품들에서 건반입력에 QKeyEvent::text()를 사용하면 서유럽이나 북미주의 느린 타자수들과 큰 차이는 없지만 text()에 의한 특수입력방식을 리용하는 빠른 타자수들이나 사람들에게는 도움이 된다.

Qt에서 사용자가 볼수 있는 문자렬인 모든 함수인수들, QLabel::setText() 및 그밖의 많은것들은 const QString &s를 가진다. QString은 다음과 같이 작업하는 const char * 로부터의 암시적강제변환을 제공한다.

```
myLabel->setText( "Hello, Dolly!" );
```

또한 다음과 같이 번역을 제공하는 함수 QObject::tr()가 있다.

```
myLabel->setText( tr("Hello, Dolly!") );
```

tr()는 const char *를 유니코드문자렬로 넘기며 설치할수 있는 QTranslator객체들을 사용하여 변환을 수행한다.

Qt는 수많은 기본 QTextCodec클라스들 즉 유니코드와 유산부호화사이의 번역방법을 알고있는 클라스들을 제공한다. 이 클라스들은 다른 프로그람들과 대화하거나 다른 파일형식으로 파일들을 읽고 써야 하는 프로그람들을 지원한다.

기정적으로 const char *에 대한 변환은 지역에 의존하는 코드를 사용한다. 그러나 응용프로그람들은 다른 지역들의 코드를 쉽게 찾아서 열린 파일이나 망련결에서 특수코드를 사용한다. 또한 기본코드들이 유지하지 않는 부호화들에 새로운 코드들을 설치할수 있게 한다. (Vietnamese/VISCII는 그러한 실례의 하나이다.)

US-ASCII와 ISO-8859-1가 아주 일반적이므로 그것들사이의 변환이 특히 빠른 함수들이 있다. 실례로 응용프로그람의 그림기호를 열기 위하여 다음과 같이 할수 있다.

```
QFile f( QString::fromLatin1("appicon.png") );
```

출력에 관해서 Qt는 유니코드로부터 체계와 서체가 제공하는 어떤 부호화로의 가장 효과적인 변환을 수행한다. 조작체계, 지역, 서체리용가능성, 사용하는 문자에 대한 Qt의 지원에 따라 이 변환은 좋을수도 있고 나쁠수도 있다. 가장 많이 사용하는 지역을 우선 강조하면서 앞으로 제공되는 판들에서 이것을 확장하려고 한다.

제8장. 이식과 가동환경

제1절. 창문체계에 고유한 알아두기

Qt는 여러 가동환경GUI도구일식이므로 대체로 전체API는 모든 가동환경과 창문체계들에서 같다. 가동환경에 고유한 특성들을 사용하여 여전히 가동환경에 의존하지않는 원천나무를 유지관리하려고 한다면 적당한 #ifdef문들을 리용하여 가동환경에 고유한 코드를 보호해야 한다.

1. Qt/X11

이 가동환경에서 콤파일할 때 마크로 Q_WS_X11 가 정의된다.

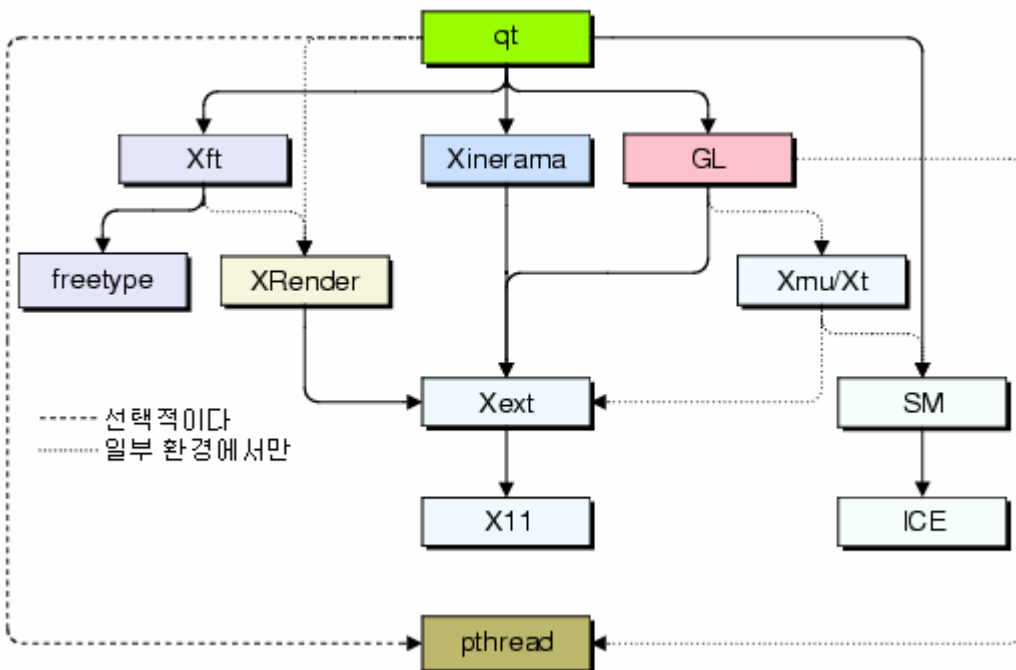



그림 8-1. Qt/X11서고의존관계

표 8-1. Qt/X11서고의존관계

이름	서고	설명
Qt	libqt 혹은 libqt-mt	Qt(libqt-mt는 다중스레드화된다)
Xft	libXft 혹은 libXft2	X Free Type Interface; -xft 혹은 자동탐지
freetype	libfreetype	FreeType;-xft 혹은 자동탐지
Xrender	libXrender	X묘사확장; -xrender 혹은 –xft 혹은 자동탐지
Xinerama	libXinerama	다중스레드지원; -xinerama 혹은 자동탐지
Xert	libXext	X Extensions
X11	libX11	X Window System
GL	libGL	OpenGL; 자동탐지(스레드화되면 –thread

이름	서고	설명
		를 요구)
Xmu/Xt	libXmu와 libXt	XX Toolkit
SM	libSM	X Session Managementl –sm
ICE	libICE	Inter-Client Exchange; -sm
pthread	libpthread	다중스레드작성; -thread (OpenGL이 스레드화되면 필요)

알아두기: 픽스매프와 화상들을 위한 알파투명도를 얻으려면 Xft와 XRender를 둘다 유지하고 콤파일해야 한다. XRender지원만 가지고 콤파일하는것으로는 충분하지 않다.

2. Qt/Windows

이 가동환경용으로 콤파일할 때 마크로 Q_WS_WIN이 정의된다.

1) Microsoft Visual Studio 2005전개문제

Microsoft Visual Studio 2005에서 콤파일한 실행파일들은 Qt와 응용프로그람자체가 의존하는 C/C++실행시서고들을 지적하는 목록파일들을 요구한다. 목록(manifest)파일은 자동적으로 생성되지만 실행가능프로그람과 같은 등록부에 배치되여야 한다.

응용프로그람을 전개할 때 목록파일을 전개한다.

알아두기: 자기 응용프로그람을 .manifest파일로 전개하려고 하지 않는다면 mt지령(Visual Studio 2005에 배포된다)에 의해 실행가능파일에 목록(manifest)을 자원으로 묻어둘수 있다.

다른 문제가 있으면 MSDN문서 "Side-by-side Assemblies Reference"를 보시오.

3. Qt/Mac OS X

이 가동환경에서 콤파일할 때 마크로 Q_WS_MACX 가 정의된다.

- Qt/Mac문제

4. Qt/Embedded

이 가동환경에서 콤파일할 때 마크로 Q_WS_QWS가 정의된다. (창문체계는 문자그대로 Qt창문체계이다.)

- 설치
- Qt/Embedded성능동조
- Qt/Embedded응용프로그람의 실행
- Qt/Embedded에로 자기 응용프로그람의 이식
- 서체형식과 정의
- 문자입력 (건반, 펜, ...)
- 위치지정장치조종 (마우스, 펜, ...)
- 기능정의파일을 가지고 기억기사용을 줄인다

제2절. Qt/Mac문제

이 파일은 Qt를 Mac OS X에서 사용할 때의 알려진 문제점들과 가능한 작업범위를 륜곽적으로 설명한다.

1. GUI 응용프로그람

GUI응용프로그람들은 widgets.app/와 같은 묶음의 밖에서 혹은 open(1)지령을 리용하여 실행하여야 한다. Mac OS X는 차림표띠에로의 호출을 얻는것은 물론 사건들을 정확히 발송하기 위하여 이것을 요구한다. GDB를 사용한다면 실행가능파일에로의 완전경로를 리용하여 실행해야 한다.

2. QCursor

Mac OS X는 16×16사용자정의유표만 가지므로 QCursor는 이것에 의하여 제한된다. 현재 이 문제와 관련한 유일한 방법은 작은 유표(16×16)를 사용하는것이다.

3. 반가명식 본문

Qt/Mac (3.0.5이상)는 Apple의 Aqua Style차림표에 따라 제안된 본문을 위한 일부 기능을 도입하였다. 이 기능은 Mac OS X >10.1.4이상으로 제한되고 이 판이 탐색되지 않으면 낡은 본문묘사서고로 넘어간다.

4. 서고유지

1) 묶음(bundle)에 기초한 서고들

동적서고들을 자기의 Mac OS X응용프로그람묶음의 부분(응용프로그람등록부)으로서 결합하려면 이것들을 응용프로그람묶음의 보조등록부인 Frameworks라는 등록부에 배치한다.

응용프로그람은 서고들이 @executable_path/../Frameworks/libname.dylib의 설치이름을 가진다면 이 동적서고들을 발견한다.

qmake와 Makefile을 사용한다면 QMAKE_LFFLAGS_SONAME설정을 사용하여야 한다.

QMAKE_LFLAGS_SONAME = -Wl,-install_name,@executable_path/../Frameworks/Project Builder의 경우에 Library목표들이 @executable_path/.../Frameworks로 설정된 자기의 설치경로(목표의 Build Settings에서)를 가지도록 설정해야 한다. 또한 SKIP_INSTALL이라고 부르는 사용자정의구축설정을 추가하고 이것을 YES로 설정해야 한다. Application목표에서 응용프로그람래퍼의 Framework보조홀더에로 서고제품을 복사하는 Copy Files구축단계를 추가해야 한다.

DYLD_LIBRARY_PATH환경변수들이 이 설정을 무시하고 /usr/lib 및 류사한 기정위치들안에서 동적서고들의 검색과 같은 다른 기정경로들로 설정된다는것을 알아야 한다.

아직도 서고코드가 Mac OS X의 2진에 포함되는 정적응용프로그람들을 구축할것을 강하게 권고한다. 그러나 플라그인기능을 요구하는 응용프로그람들을 싣는 경우에 동적서고들을 자기 응용프로그람의 부분으로 사용하여야 한다.

2) 서고들의 결합

Qt 3.1동적서고들을 결합하는 새로운 동적서고를 구축하려고 한다면 ld -r기발을 받아들여 재배치정보를 출력파일에 보관함으로써 이 파일이 다른 ld실행의 주제로 되게 할수 있다. 이것은 .pro파일에 -r기발을 설정하고 LFLAGS설정에 의해 수행된다.

3) 초기화순서

dyld(1)은 대역정적초기화자들을 자기의 응용프로그람에 련결하기 위하여 그것들을 호출한다. 서고가 Qt에 련결되고 Qt에서(자체의 서고에서 대역초기화자들로부터) 대

역변수들을 참고한다면 자기의 서고보다 먼저 Qt와 련결하는가 확인해야 하며 그렇지 않으면 Qt의 대역초기화자들이 아직 호출되지 않았으므로 결과는 정의되지 않는다.

4) 플라그인 기능

Project Builder 혹은 Xcode를 리용하여 Qt플라그인들을 구축할수 없다. qmake에 의해 환경을 구성하고 플라그인들을 구축한다.

5. 콤파일러설정

- 콤파일시기발

정의에 특정한 Mac OS X코드를 포함하려고 한다면 다음과 같이 Q_OS_MACX기발을 사용한다.

```
#if defined(Q_OS_MACX)
// the code used
#endif
```

Mac OS X 10.2하에서 구축할 때 MACOSX_102기발이 자동적으로 make구축에 포함된다.

6. Qt/Mac의 구축 및 환경구성

- 정적환경구성문제

정적서고가 designer make단계에서 다음과 같은 오유로 실패하면

QWidget::sizeHint() const referenced from libqui expected to be defined in @executable_path/../Frameworks/libqt-mt.3.dylib non-virtual thunk [nv:-40] to QWidget::metric(int) const referenced from libqui expected to be defined in @executable_path/../Frameworks/libqt-mt.3.dylib

자기의 서고경로가 libqui서고들 혹은 기호련결을 가지지 않는가 확인한다. 이것들을 삭제하면 구축은 계속된다.

7. Macintosh Native API호출

1) 묶음경로의 호출

Macintosh응용프로그람은 실제로 .app로 끝나는 등록부이다. 이 등록부는 여러가지 다른 보조등록부들과 원천을 포함한다. 이 묶음안에 실례로 플라그인등록부를 배치하려고 한다면 묶음이 디스크의 어디에 상주하는지 알아야 한다. 다음의 코드는 이것을 수행한다.

```
CFURLRef pluginRef =
CFBundleCopyBundleURL(CFBundleGetMainBundle());
CFStringRef macPath = CFURLCopyFileSystemPath(pluginRef,
   kCFURLPOSIXPathStyle);
const char *pathPtr = CFStringGetCStringPtr(macPath,
   CFStringGetSystemEncoding());
qDebug("Path = %s", pathPtr);
CFRelease(pluginRef);
CFRelease(macPath);
```

#if defined(Q_OS_MACX)마크로명령문안에 이것을 넣어야 한다.

2) 응용프로그람 차림표와 원시대화칸의 번역

지역화된 응용프로그람 차림표와 원시대화칸을 얻으려면 약간한 작업을 해야 한다. 이것은 Mac OS X가 요구하는것이지 Qt의 요구가 아니다.

우선 지역화된 자원홀더를 Bundle안에 추가해야 한다.

http://developer.apple.com/documentation/CoreFoundation/Conceptual/

CFBundles/index.html

그리고 제목Adding Localized Resources를 찾는다.

중요하게 할 일은 locversion.plist라는 파일을 창조하는것이다. 여기에 노르웨이어의 실례가 있다.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>LprojCompatibleVersion</key>
  <string>123</string>
  <key>LprojLocale</key>
  <string>no</string>
  <key>LprojRevisionLevel</key>
  <string>1</string>
  <key>LprojVersion</key>
  <string>123</string>
</dict>
</plist>
```

그다음 자기가 요구하는 언어로 응용프로그람을 실행할 때 노르웨이어로 설정하여 "Quit"대신에 "Avslutt"와 같은 차림표항목을 표시한다.

8. 사용자대면부

1) 오른쪽마우스찰칵

Mac OS X용으로 오른쪽마우스찰칵기능을 제공하려면 QContextMenuEvent클라스를 사용한다. 이것은 상황차림표 다시 말하여 올리펼침선택을 현시하는 차림표로 변환된다. 이것은 오른쪽마우스찰칵의 가장 일반적인 사용이고 조종건-찰칵을 Mac OS X의 1단추마우스기능으로 변환한다.

2) 차림표띠

Qt/Mac는 자동적으로 자기의 차림표띠들을 탐지하고 그것들을 Mac원시차림표띠들로 변환한다. 이것을 자기의 현존 Qt응용프로그람에 적합하게 조절하는 조작은 보통 자동적으로 진행되지만 특별한 요구가 있다면 현재 Qt/Mac는 능동창문(즉 QApplication::activeWindow())으로부터 시작하여 차림표띠를 선택하고 다음과 같이 적용한다.

① 창문이 QMenuBar를 가지면 그것이 사용된다.

② 창문이 이행금지이면 차림표띠가 사용된다. 차림표띠가 지정되지 않으면 기정차림표띠가 아래에 서술하는것처럼 사용된다.

③ 창문이 부모를 가지지 않으면 기정차림표띠가 아래에 서술하는것처럼 사용된다.

우의 세 걸음은 우의 하나가 만족될 때까지 부모창문사슬우의 모든 길에 적용된다. 모두가 실패하면 기정차림표띠가 창조되며 Qt/Mac에서 기정차림표띠는 빈 차림표띠이지만 부모없이 QMenuBar를 창조함으로써 각이한 기정차림표띠를 창조할수 있으며 처음 창조된것은 기정차림표띠로 지정되고 기정차림표띠가 요구될 때마다 사용된다.

9. 제한

1) 차림표항목

• QCustomMenuItems은 Mac원시차림표띠들에 유지되지 않고 Mac원시차림표띠

에 없는 올리펼침차림표들에 유지된다.

• 하나이상의 건치기(QKeySequence)를 가지는 지름건을 가진 항목들이 있으면 처음건이 사용된다.

2) 유지되지 않은 원시창문부품들

현재 Qt/Mac는 속성지(sheet), 그리기기구(drawer) 혹은 선택기(chooser, Panther형식타브들)를 유지하지 않는다. Qt의 다음 판들에서 이것들을 유지할수 있다.

제9장. Qt/Embedded

제1절. Qt/Embedded의 설치

설치수속은 Linux용으로 작성되고 다른 가동환경에서는 수정해야 한다.

① 이미 수행하지 않았으면 압축파일의 압축을 푼다.

cd <anywhere>
gunzip qt-embedded-commercial-VERSION.tar.gz # uncompress the archive tar xf qt-embedded-commercial-VERSION.tar # unpack it

VERSION을 Qt/Embedded판번호로 완전히 교체한다.

여기서는 압축파일가 ~/qt-embedded-commercial-VERSION으로서 설치된다는 것을 가정한다.

② Qt/Embedded서고와 실례들을 콤파일한다.

```
cd ~/qt-embedded-commercial-VERSION
export QTDIR=~/qt-embedded-commercial-VERSION
./configure
make
```

환경구성체계는 가동환경에 고유한 선택들을 추가하도록 설계되지만 일반적으로 틀완충기를 유지하는 모든 Linux체계는 "linux-generic-g++"가동환경을 사용할수 있다. 또한 환경구성체계는 교차콤파일러를 유지하며 Linux/x86에서 Linux/MIPSEL목표에 대하여 구축하려면 다음과 같이 리용할수 있다.

```
./configure -embedded mips
```

적은 수의 환경구성만 미리 정의된다. mkspecs/qws/등록부에 새 파일들을 추가하여 자기의 사용자정의환경구성을 창조할수 있다. 출발점으로서 류사한 현존환경구성을 사용한다.

알아두기: 환경구성스크립트에서 오유로 인하여 리틀엔디안콤퓨터(실례로 x86)에서 비그엔디안처리소자(실례로 PowerPC)용으로 교차콤파일할 때 목표대신에 주콤퓨터의 엔디안성을 사용한다. 주의: configure를 실행한 후에 make를 실행하기전에 $QTDIR/include/qconfig.h를 편집하고 Q_BYTE_ORDER의 정의를 변경한다.

③ 틀완충기유지를 허용한다.

틀완충기를 허용하도록 핵심을 재콤파일할 필요가 있다. 여기서는 그 수행방법을 서술하지 않으며 틀완충기HOWTO페지에는 간단한 설명이 있다. (틀완충기를 허용하면 기동시에 펭긴새로고가 보인다.)

Matrox G100/G200/G400에서는 matrox틀완충기구동프로그람을 사용한다.

NVidia TNT카드에서는 nvidia틀완충기구동프로그람을 사용한다.

Mach64와 대다수 다른 카드들에서는 vesafb구동프로그람을 사용한다.

일부 카드는 VGA16방식에서만 유지되며 이것은 현재 판의 Qt/Embedded에서는 VGA/16이 아직 유지되지 않았으므로 작업하지 않는다. 자기 핵심을 갱신하거나 지어는 실험적핵심으로 절환할 필요가 있다.

또한 틀완충기는 boot파라메터에 의해 허용되여야 한다. (자세한것은 /usr/src/linux/Documentation/fb를 참고하시오.)

fbset프로그람은 Linux배포물에 포함되며 체계를 재기동하지 않고 비데오방식을 절환하는데 쓰인다. 봉사기가 기동할 때 능동으로 되는 비데오방식이 사용된다. (8bit방식들은 여전히 실험적이다.)

알아두기: fbset는 vesafb구동프로그람과 작업하지 않는다.

④ 허가를 변경한다.

Qt/Embedded를 실행하려면 틀완충장치 /dev/fb0.에 대한 호출을 쓸 필요가 있다.

또한 마우스장치에 대한 호출을 읽어들여야 한다. (보통 /dev/mouse가 기호련결이고 실제의 마우스장치는 읽기가능이여야 한다.)

⑤ 실례프로그람을 실행하는 방법

다음과 같이 가상콘솔로 가입하고 수행한다.

```
cd ~/qt-embedded-commercial-VERSION/examples/launcher
./start-demo
```

⑥ 여러가지 난관을 알리는 오유들

gpm를 없애려면 루트로서 다음의 지령을 실행한다.

```
gpm -k
```

일부 경우에 봉사기가 작업하지 않으면 루트로서 실행할 때 그것이 작업한다.

틀완충기를 사용하여 프로쎄스들을 보여준다.

```
fuser -v /dev/fb0
```

그러한 프로쎄스들을 없앤다.

```
fuser -vk /dev/fb0
```

혹은 거슬리는것을 없앤다.

```
fuser -k -KILL /dev/fb0
```

현존 쎄마퍼를 보여준다.

```
ipcs
```

쎄마퍼들을 삭제한다.

```
ipcrm
```

의뢰기와 봉사기사이의 통신은 이름있는 파이프 /tmp/qtembedded-username/QtEmbedded-0를 통하여 수행되고 흔히 그것을 삭제하는데(실례로 루트특권으로 Qt/Embedded를 실행한다면 후에 특권없는 사용자로서) 필요할수 있다.

⑦ 전용화

Qt/Embedded서고는 불필요한 특성의 삭제에 의하여 크기를 줄일수 있다.

⑧ 개발과 오유수정을 위하여 대신에 Qt/Embedded가상틀완충기를 사용하는것이 더 간단하다.

제2절. 특성정의파일

src/tools로부터 환경파일 qconfig.h를 수정함으로써 설치에서 사용하는 완전한 Qt기능의 부분모임을 정의할수 있다. 환경에 대한 –qconfig선택은 환경구성을 선택하는데 사용된다.

이러한 수정은 Qt의 크기를 줄이는것이 중요하고 응용프로그람설정이 자주 고정되는 Qt/Embedded가동환경에서만 리용된다.

qconfig.h정의파일은 특성들을 허용하지 않도록 마크로들을 정의한다. 일부 특성들은 다른 특성에 의존하며 이 의존관계는 qfeatures.h에서 표시된다.

유효선택은 표 9-1과 같다.

표 9-1. 유효선택

마크로	금지	다음에 의하여 자동 설정
화상 (QImageIO)		
QT_NO_IMAGEIO_BMP	Microsoft Bitmap화상파일형식.	

마크로	금지	다음에 의하여 자동 설정
QT_NO_IMAGEIO_PPM	Portable Pixmap화상파일형식.	
QT_NO_IMAGEIO_XBM	X11 Bitmap화상파일형식.	
QT_NO_IMAGEIO_XPM	X11 Pixmap화상파일형식.	
QT_NO_IMAGEIO_PNG	Portable Network Graphics화상파일형식.	
동화상		
QT_NO_ASYNC_IO	비동기I/O (QAsyncIO)	
QT_NO_ASYNC_IMAGE_IO	비동기화상I/O과 GIF화상지원 (QImageDecoder, ...)	
QT_NO_MOVIE	동화지원 (QMovie)	QT_NO_ASYNC_IO, QT_NO_ASYNC_IMAGE_IO
서체		
QT_NO_TRUETYPE	TrueType (TTF와 TTC)서체파일형식. Qt/Embedded에서만 사용.	
QT_NO_BDF	Bitmap Distribution Format (BDF)서체파일형식. Qt/Embedded에서만 사용.	
QT_NO_FONTDATABASE	서체자료기지.	
국제화		
QT_NO_I18N	유니코드와 8-bit부호화사이의 변환.	
QT_NO_UNICODETABLES	모든 유니코드문자 용 대소문자변환과 같은것을 정의하는 대규모표.	
MIME		
QT_NO_MIME	형있는 자료(실례로 본문, 화상, 색)의 부호화와 타그화용 Internet 표준인 Multipurpose Internet Mail Extensions (QMimeSource)	
QT_NO_RICHTEXT	HTML풍의 본문 (QStyleSheet, QLabel)	QT_NO_MIME
QT_NO_DRAGANDDROP	응용프로그람들사이의 끌어다놓기자료 (QDragObject)	QT_NO_MIME
QT_NO_CLIPBOARD	응용프로그람들사이의 자르기 및 붙이기 (QClipboard)	QT_NO_MIME
음성		
QT_NO_SOUND	음성파일재생 (QSound)	
스크립팅		
QT_NO_PROPERTIES	Qt기초의 응용프로그람의 스크립팅.	

마크로	금지	다음에 의하여 자동 설정
Qt/Embedded에 고유		
QT_NO_QWS_CURSOR	Qt/Embedded에서 유표sprite. 펜 조작장치는 보통 이 특성을 요구하지 않는다.	
QT_NO_QWS_DEPTH_8GRAYSCALE	화소당 8bit: 재색의 256준위. QWS_DEPTH_8와 비호환.	
QT_NO_QWS_DEPTH_8	화소당 8bit: 40보조색을 가지는 216색립방체. QWS_DEPTH_8GRAYSCALE와 비호환.	
QT_NO_QWS_DEPTH_15	화소당 15bit: 적,록,청의 매개에 32준위.	
QT_NO_QWS_DEPTH_16	화소당 16bit: 록에 64준위, 적과 청색에 32준위씩.	
QT_NO_QWS_DEPTH_32	화소당 32bit: 적, 록, 청 각각에 256준위.	
QT_NO_QWS_MACH64	Mach64가속구동기 (시위뿐).	
QT_NO_QWS_VFB	X11에서 가상틀완충기실행 (참고문서를 보시오).	
망프로그람작성		
QT_NO_NETWORK PROTOCOL	국부파일검색을 포함하는 추상다중통신규약자료검색 (QNetworkProtocol)	
QT_NO_NETWORKPRO TOCOL_FTP	FTP통신규약자료검색.	QT_NO_NETWORK PROTOCOL
QT_NO_NETWORKPRO TOCOL_HTTP	HTTP통신규약자료검색.	QT_NO_NETWORK PROTOCOL
그리기와 그림		
QT_NO_COLORNAMES	일부 QColor구성자들과 일부 HTML문서(QColor, QStyleSheet) 들에서 쓰이는 "red"와 같은 색이름들	
QT_NO_TRANSFORMATIONS	Qt에서 많은 클라스들에서 사용한다. 이와 함께 회전과 비례화도 가능하다. 이것이 없으면 좌표변환(QWMatrix)만 할수 있다.	
QT_NO_PSPRINTER	PostScript 인쇄기지원.	
QT_NO_PRINTER	인쇄기지원(QPrinter)	QT_NO_PSPRINTER (Unix only)
QT_NO_PICTURE	Qt그리기지령들을 파일들에 보관한다. (QPicture)	

마크로	금지	다음에 의하여 자동 설정
창문부품		
QT_NO_WIDGETS	이것을 허용하지 않으면 QWidget를 제외한 모든 창문부품들을 금지한다.	
QT_NO_TEXTVIEW	HTML문서보기 (QTextView)	QT_NO_WIDGETS, QT_NO_RICHTEXT
QT_NO_TEXTBROWSER	HTML문서열람(QTextBrowser)	QT_NO_TEXTVIEW
QT_NO_ICONVIEW	표식화된 그림기호들 (QIconView)	QT_NO_WIDGETS, QT_NO_DRAGANDDROP
QT_NO_LISTVIEW	정보목록 (QListView)	QT_NO_WIDGETS
QT_NO_CANVAS	객체캔버스 (QCanvas)	QT_NO_WIDGETS
QT_NO_DIAL	값조종 (QDial)	QT_NO_WIDGETS
QT_NO_WORKSPACE	MDI (다중문서대면부) (QWorkspace)	QT_NO_WIDGETS
QT_NO_LCDNUMBER	LCD형식의 수값현시 (QLCDNumber)	QT_NO_WIDGETS
GUI형식		
QT_NO_STYLE_ WINDOWS	Microsoft Windows형식(QWindowsStyle)	QT_NO_WIDGETS
QT_NO_STYLE_MOTIF	OSF Motif형식(QMotifStyle)	QT_NO_WIDGETS
QT_NO_STYLE_CDE	Open Group CDE형식(QCDEStyle)	QT_NO_STYLE_MOTIF
QT_NO_STYLE_AQUA	MacOS X형식(QAquaStyle)	
QT_NO_STYLE_ PLATINUM	MacOS 9형식(QPlatinumStyle)	QT_NO_WIDGETS
QT_NO_STYLE_SGI	SGI형식(QSGIStyle)	QT_NO_STYLE_MOTIF
대화칸		
QT_NO_DIALOGS	이것을 허용하지 않으면 모든 공통대화칸 QWidget들을 금지한다.	QT_NO_WIDGETS
QT_NO_FILEDIALOG	파일선택대화칸 (QFileDialog)	QT_NO_DIALOGS, QT_NO_NETWORKPROTOCOL, QT_NO_LISTVIEW
QT_NO_FONTDIALOG	서체선택대화칸 (QFontDialog)	QT_NO_DIALOG

마크로	금지	다음에 의하여 자동 설정
		S, QT_NO_FONT DATABASE
QT_NO_COLORDIALOG	색선택대화칸 (QColorDialog)	QT_NO_DIALOGS
QT_NO_INPUTDIALOG	본문입력대화칸 (QInputDialog)	QT_NO_DIALOGS
QT_NO_MESSAGEBOX	통보문/재촉문대화칸 (QMessageBox)	QT_NO_DIALOGS
QT_NO_PROGRESS DIALOG	긴계산진척상황대화칸 (QProgressDialog)	QT_NO_DIALOGS
QT_NO_TABDIALOG	타브페지대화칸 (QTabDialog)	QT_NO_DIALOGS
QT_NO_WIZARD	여러걸음대화칸 (QWizard)	QT_NO_DIALOGS

제3절. Qt/Embedded에 고유한 클라스들

Qt/Embedded클라스들은 두가지로 분류된다. 즉 기본은 매개 Qt/Embedded프로그람에서 사용되며 일부는 Qt/Embedded봉사기에 의해서만 사용된다. 또한 Qt/Embedded봉사기프로그람은 단일프로쎄스설치인 경우에 의뢰기일수도 있다. Qt/Embedded에 고유한 모든 원천파일들은 src/kernel에 있으며 뒤붙이 qws가 있다. -기호는 계승을 가리킨다.

① QFontManager

응용프로그람마다 다음것들중 하나가 있다. 응용프로그람이 기동할 때 $QTDIR/etc/fonts/fontdir (혹은 QTDIR가 정의되지 않으면 /usr/local/etc/qt-embedded/fonts/fontdir)로부터 서체정의파일을 읽어들인다. 이것은 모든 서체정보를 유지하며 표시된 서체들의 캐쉬를 관리한다. 또한 서체공장을 창조하며 QFontManager::QFontManager는 새 공장용구성자를 추가하는 곳이다. 이것은 특정한 서체를 요구하기 위한 고급한 대면부를 주며 QFontFactories를 호출하여 요구에 따라 디스크로부터 서체들을 적재한다. 이것은 오직 BDF와 TrueType서체들에만 적용되며 Qt/Embedded의 최적화된 .qpf서체파일형식은 QFontManager기구와 함께 넘긴다.

서체대조 혹은 동작캐싱을 변경하지 않으려면 이 클라스를 수정하지 말아야 한다.

② QDiskFont

이것은 하나의 디스크서체파일정보를 포함한다. (실례로 /usr/local/etc/qt-embedded/times.ttf). 이것은 파일경로, 서체가 비례가능한가하는것과 그 무게, 크기, Qt/Embedded이름 등에 대한 정보를 보관한다. 이 정보를 리용하여 QFontManager는 제일 근사한 디스크서체를 찾을수 있다. (이것은 일치하는 이름들에 대하여 무게를 정한 득점기록기구를 사용하고 그다음에는 서체가 경사체인가, 아닌가, 그다음 그 무게를 리용한다.)

이 클라스를 수정할 리유는 없다.

③ QRenderedFont

체계에 의하여 현재 적재된 매개의 유일한 서체에 대하여 오직 하나의 QRenderedFont가 있다. (즉 매개가 이름, 크기, 무게, 경사체의 유일한 결합). QRenderedFont들은 참고계수되며 QRenderedFont를 사용하는것이 없으면 글리프비트매프의 캐쉬와 함께 삭제된다. 적재된 QDiskFont는 그 QFontFactory에 의하여 열려져있다.

글리프가 캐쉬되는 방법을 변경하지 않는다면 이 클라스를 수정할 까닭이 없다.

④ QFontFactory와 자손들인 QFontFactoryBDF, QFontFactoryTtf

이것들은 특정한 서체형식 실례로 비례가능한 Truetype와 Type1형식(둘다 Freetype 2을 사용하는 QFontFactoryTtf에서 유지) 그리고 X에서 사용하는 비트매프BDF형식에 대한 지원을 제공한다. 이것은 디스크서체를 열기 위하여 호출된다. 서체가 열리면 디스크로부터 새 서체의 창조를 고속화하기 위하여 열려진채로 있다. 또한 QRenderedFont를 창조하고 유니코드값들로부터 서체파일에로의 첨수로 변환할수 있다. 단순성을 위하여 글리프는 유니코드순서가 아니라 서체에 정의되는 순서와 첨수로 보관된다.

이 클라스를 수정할 필요가 없지만 각이한 형태의 서체묘사기를 추가하려고 한다면 (실례로 사용자정의벡토르서체형식에 대하여) 계승되여야 한다.

⑤ QGlyph

이것은 QRenderedFont로부터 문자의 특별한 화상을 서술한다. 실례로 Times New Roman, 강조경사체, 반별명식의 10point의 문자 'A'. 이것은 그 문자에 대한 정보를 가진 QGlyphMetrics구조체의 지적자와 그 글리프에 대한 생자료의 지적자를 포함한다. 이것은 1bit마스크이거나 8bit알파통로이다. 매개의 QRenderedFont는 요구에 따라 이것들을 창조하고 보관한다. (이것은 현재 TrueType서체들에 실현되지 않는다.)

가령 본문서체를 유지하기 위하여 Qt/Embedded를 수정하고있다면 이 클라스를 수정할 필요가 있는데 이 경우에 QGfxRaster도 수정하여야 한다.

⑥ QMemoryManagerPixmap/QMemoryManager

이 처리함수들은 픽스매프들을 위한 공간을 요구하며 또한 QPF형식서체들의 궤적을 보유한다. (이것들은 크기가 2-20KB정도인 QRenderedFonts의 자그마한 상태출력이며 기억기에 보관하기 위하여 디스크로부터 직접 mmap될수 있다.) 새로운 QRenderedFont를 창조할 필요는 없고 서체요구와 일치하는 QPF서체가 발견된다. 자기의 서체요구가 적당하면(실례로 고정점크기를 적게 요구한다면) 모든 QFontFactory기능을 버리고 간단히 QPF들을 사용할수 있다. 제일 잘 일치하는 적재가 QPF들에서 수행되지 않으면 QFontManager를 거쳐서 적재되는것들과 반대이므로 점크기와 꼭 같은 QPF가 없으면 그 크기의 본문이 간단히 표시되지 않는다.

이 클라스를 수정할 필요는 없다.

⑦ QScreen→QLinuxFbScreen→가속화된 화면들, QTransformedScreen→QVfbScreen

이것들은 Qt/Embedded가 그리고있는 틀완충기를 은페하며 틀완충기를 회전하기 위한 자리표변환을 유지하며 조색판의 실현을 가능하게 하고 개별적인 틀완충기억기들을 가지는 장치들에 대하여 비화면그라픽스에로의 호출을 제공한다.

이것은 픽스매프캐쉬과 가속화된 픽스매프=>화면blt를 가능하게 하는데 쓰인다. QLinuxFbScreen과 가속화된 화면은 Linux /dev/fb대면부를 사용하여 그라픽스기억기에 대한 호출과 장치의 특성에 대한 정보를 얻는다. 열려는 틀완충기는 QWS_DISPLAY에 의하여 지정된다. 유일한 QTransformedScreen은 회전식 틀완충기에 대한 지원을 실현한다. QVfbScreen는 모의식 틀완충기를 포함하는 X창문을 제공한다. (공유기억기덩어리는 틀완충기결에 설정되여 X창문에 blt된다.) 이것은 X를 완료

함이 없이 Qt/Embedded하에서 사용자들이 자기의 응용프로그람들을 수정하게 하는 오유수정장치로서 사용하는 경향이 있다. 가속된 화면구동기들은 그것들이 QWS_CARD_SLOT에 의하여 지정된 장치(지정되지 않으면 기정으로 AGP처리부의 일반위치로 되여있다)를 구동할수 있는가 검사하고 /dev/mem으로부터 소편상등록기들을 사영한다. 또한 그것들은 알려진 값들로 등록기들을 초기화하여 소편에 고유한 설정을 진행할수 있다. 끝으로 QScreen은 새로운 QScreenCursor와 QGfx들을 창조하는데 리용된다.

픽스매프들이 기억기에 할당되는 방법을 수정하려고 한다면 QLinuxFbScreen의 파생클라스를 만들거나 수정한다. 가속화된 장치구동프로그람을 쓰고있다면 QScreen 혹은 QLinuxFbScreen의 파생클라스를 만들 필요가 있다.

⑧ QScreenCursor→가속화된 유표→QVfbCursor

이것은 화면상의 마우스유표그리기와 비가속화된 유표형에 대하여 화면의 보존과 되살리기를 조종한다.

QScreenCursor의 파생클라스작성은 가속화된 구동기에 최적이다. (하드웨어가 하드웨어유표를 유지하면 그렇게만 하려고 할수 있다.)

⑨QGfx→RasterBase→Raster→가속화된 구동기→QGfxVfb→QGfxTransformedRaster

이 클라스는 저수준 QPainter와 같은 그리기조작을 은페한다. QGfxRaster와 그 자손들은 미숙한 틀완충기에로 그리기 위하여 특별히 작성되였다. 이것들은 그리기조작을 위한 변위와 창문에 대한 그리기를 지원하기 위한 절단령역을 가질수 있다. 가속화된 구동기를 실현하기 위하여 QGfxRaster형판의 파생클라스를 만들어야 한다.

QGfxRaster를 수정하여 그리기방법을 전용화하거나 새로운 비트깊이와 화소형식의 지원을 추가할수 있다.

⑩ QLock, QLockHolder

이것은 Qt/Embedded의뢰기들사이에 공유된 기억기에 대한 동기호출에 쓰이는 System V신호기발(semaphore)을 은페한다. QLockHolder는 QLock를 더 간단히 관리하고 해체하게 하는 편의클라스이다.

System V IPC없는 조작체계에 Qt/Embedded를 이식하지 않는다면 이 클라스를 수정할 필요가 없다.

⑪ QDirectPainter

이것은 가리키고있는 창문의 지적자와 창문의 절단령역 등을 주는 QPainter이다. 이것은 창문내용의 화소준위조작을 간단히 수행하기 위한것이다.

이 클라스를 수정할 리유는 없다.

⑫ QWSSoundServer, Client

Qt/Embedded봉사기는 간단한 음성재생기와 혼합기를 포함한다. 의뢰기들은 파일로서 지정된 봉사기재생음성을 요구할수 있다.

Linux형식의 /dev/dsp없는 조작체계에 Qt/Embedded를 이식하지 않는다면 이 클라스를 수정할 필요가 없어야 한다.

⑬ QWSWindow

이것은 개별적인 제일 웃준위창문의 봉사기능력을 포함한다. 그것이 할당되는 틀완충기의 령역, 그것을 창조한 의뢰기 등.

이 클라스를 수정할 리유는 없다.

⑭ QWSKeyboardHandler→보조형

이것은 건반/단추입력을 조종한다. QWSKeyboardHandler는 파생클라스로서 /dev/tty와 독단적인 저준위USB사건장치(USB건반에 대하여), 일부 PDA단추장치들을 읽기 위하여 제공된다.

QWSKeyboardHandler의 수정은 각이한 형식의 건반을 유지하게 하며 (현재 상

당히 표준인 US PC형식건반만 유지된다.) 그 파생클라스를 만드는것은 비지적자입력장치들을 조종하는 좋은 방법이다.

⑮ QWSMouseHandler→QWSCalibratedMouseHandler→마우스형

이것은 마우스/손접촉화면입력을 조종한다. QWSCalibratedMouseHandler의 자손들은 려파코드를 사용하여 손접촉화면에서 지적자의 순간요동(jittering)을 방지한다. 일부 매몰형 장치는 핵심에서 이 려파를 수행하며 그 경우에 구동기는 QWSCalibratedMouseHandler로부터 계승할 필요가 없다.

QWSCalibratedMouseHandler의 파생클라스작성은 핵심려파없는 손접촉화면에 선택되며 QWSMouseHandler의 계승은 다른 형의 지정장치(펜타블레트, 손접촉화면, 마우스, 추적구 등)를 추가하는 방법이다.

⑯ QWSDisplay

이 클라스는 Qt/Embedded봉사기에서만 존재하며 체계에서 건반과 마우스는 물론 모든 제일웃준위창문들의 궤적을 보유한다.

Qt/Embedded창문(실례로 알파상표가 있는 창문들)동작에 대한 심중한 수정을 하려면 이것만 수정하면 된다.

⑰ QWSServer

이것은 의뢰기에 대한 Qt/Embedded봉사기의 Unix령역소케트련결을 관리한다. 이것은 QWS통신규약사건들을 송수신하며 QWSDisplay를 호출하여 창문의 할당령역을 변경하는 일을 수행한다.

이것을 수정하려는 유일한 리유는 소케트부류의 기구가 아닌 다른것을 리용하여 Qt/Embedded응용프로그람들사이에 교제하려는것이다. (그러한 경우에 QWSClient도 수정한다.) Unix령역소케트와 같은것을 가지고있으면 그대신에 QWSSocket/QWSServerSocket를 수정한다. 응용프로그람들사이에 교제하기 위하여 여분의 QWS사건들을 추가하지 말고 대신에 QCOP를 사용한다.

⑱ QWSClient

이것은 Qt/Embedded련결의 의뢰기측면을 은폐하며 사건들을 정렬하고 해체할수 있다. Qt/Embedded응용프로그람들사이에 교제하기 위하여 Unix령역소케트와 근본적으로 다른것을 사용하는 경우를 제외하고 이것을 수정할 리유는 없다.

⑲ QWSDisplayData

이것은 QWS봉사기로부터 오는 사건들을 읽어들이고 해석하는 의뢰기의 QWSClient를 관리한다. 이것은 응용프로그람기동시에 QWS봉사기와 련결하여 틀완충기에 대한 정보를 얻고 기억관리기를 창조한다. 틀완충기에 대한 다른 정보는 QLinuxFbScreen의 /dev/fb로부터 직접 온다.

이것을 수정할 리유는 없다.

⑳ QWSCommand들

이것들은 QWS봉사기에 대하여 송신하는 자료를 은폐한다.

이것들을 수정할 리유는 없다.

㉑ QCopChannel

QCop는 Qt/Embedded응용프로그람들사이에 교제를 위한 간단한 IPC기구이다. 선택적인 2진자료를 가지는 문자렬통보는 각이한 통로들에 전송될수 있다.

그 기구자체는 사용자가 제일 우에 있는 기구를 구축하는것이 기본으로 되도록 설계된다.

㉒ QWSManager

이것은 제목띠의 그리기와 창문크기조절에 대한 사용자요구의 처리 등 Qt/Embedded창문관리를 제공한다.

이것을 수정할 리유는 없지만 창문동작을 수정하려면 그 파생클라스를 만들어야 한다.

㉓ QWSDecoration

이 클라스의 자손들은 Qt/Embedded창문관리기에 대하여 형식이 각이하다. 실례로 QWSWindowsDecoration은 Windows CE의 형식으로 Qt/Embedded창문틀을 그린다.

그로부터 클라스를 파생시켜 Windows XP 혹은 Enlightenment주제와 등가한 새로운 창문관리기의 외관을 제공한다.

㉔ QWSPropertyManager

이것은 QWS의뢰기의 대면부를 QWS속성체계(류사한 판의 X속성체계. 이것은 제일 웃준위창문들에 옹근수로 식별된 독단적인 자료를 붙이게 한다.)에 제공한다.

이것을 수정할 리유는 없다.

㉕ QWSRegionManager

의뢰기와 봉사기에서 제일 웃준위창문령역의 관리를 방조한다.

이것을 수정할 리유는 없다.

㉖ QWSSocket, QWSServerSocket

Unix령역소케트를 제공한다.

비Unix OS에 이식하고있는데 Unix령역소케트(바이트지향의 믿음성있고 순서화된 전송기구. 물론 통보대기렬과 같은것을 가지고도 실현할수 있다.)와 류사한것을 가지려면 이것을 수정한다.

제4절. Qt/Embedded에 가속기그라픽스구동프로그람추가

Qt/Embedded는 하드웨어가속기를 사용할수 있는 능력을 가지고있다. PCI 혹은 AGP구동기용 하드웨어가속기를 사용하려면 다음의 단계들을 수행해야 한다.

① QLinuxFbScreen의 가속기자손을 정의한다.

이것은 등록기들을 변환하기 위하여 QVoodooScreen::connect()를 실현한다. qt_probe_bus를 사용하여 PCI환경공간의 지적자를 얻는다. 이것은 PCI장치/제작회사 ID정보를 사용하여 정확한 장치를 가리키고있는가를 검사하는 곳이다. 그다음 PCI환경공간을 리용하여 자기 장치의 가속기등록기들을 물리기억기에 배치하고 /dev/mem으로부터 적당한 령역을 사영한다. 프레임완충기를 사영할 필요는 없으며 QLinuxFbScreen가 이것을 수행한다. 문제가 생기면 FALSE를 돌려준다. QVoodooScreen::initDevice()는 QWS봉사기에 의해서만 호출되며 그리기가 수행되기전에 호출되도록 담보된다. (그리하여 알려진 상태들로 등록기들을 설정할수 있는 위치이다.) connect()는 매개의 련결되는 의뢰기에 의하여 호출된다.

② QGfxRaster의 가속기자손을 정의한다.

이것은 실제의 그리기코드가 이행하는 곳이다. 하드웨어에서 실현되지 않은것은 QGfxRaster에로 넘기여 쏘프트웨어에서 수행할수 있다. optype변수를 리용하여 가속조작과 비가속조작들이 동기화된다는것을 확인한다. (하드웨어가속기가 아직 그리고있는 령역에로 쏘프트웨어를 거쳐서 그리기를 시작하면 그리기조작은 잘못된 순서로 나타난다.) optype는 공유기억기에 보관되고 비가속조작에 의하여 0으로 설정되며 가속조작은 1로 설정된다. 쏘프트웨어그라픽스조작이 요구되고 optype가 1일 때 QGfxRaster::sync()가 호출되고 그라픽스엔진에 일감이 없기를 기다리는 자체의 실현을 제공해야 한다. 역시 lastop는 최적화에 사용되고 공유기억기에 보관되며 이것은 쏘프트웨어전용QGfx에 의하여 설정되지 않으며 자기의 마지막 조작의 형(실례로 직4각형그리기)을 보관하여 같은 조작이 순차로 많이 수행될 때 다음 조작을 위한 설정부분을 피하는데 리용할수 있다.

모든 그리기조작은 임의의 등록기, lastop 혹은 optype가 호출되기 전에 QWSDisplay::grab()를 통하여 보호되여야 하며 마감에 ungrabbed()해야 한다. 이것은 동시에 두개의 응용프로그람들이 가속기를 호출하는것을 방지하며 될수록 콤퓨터를 잠그어둔다. 이것은 자기의 원천코드가 그라픽스카드에 놓이지 않을수 있게 하므로 그러한 경우를 검사하고 필요하다면 쏘프트웨어로 되돌아와야 한다. QGfxRaster는 QPainter가 직접 유지하지 않는 기능을 유지한다.(실례로32bit의 알파통로와 stretchBlt). 이 기능들은 Qt에 의해 사용되며 stretchBlt는 QPixmap::xForm()와 drawPixmap()를 변환된 QPainter로 가속화하며 알파통로가속은 32bit픽스매프를 위하여 유지된다.

③필요하다면 QScreenCursor의 가속기자손들을 정의한다. restoreUnder()와 saveUnder(), drawCursor(),draw()는 null조작으로 정의되여야 한다. set()와 move(), show(), hide()를 실현한다. 4KB는 틀완충기의 보임부분의 끝에 자기 유표용으로 남겨둔다. (즉 (width*height*depth)/8에)

④자기의 QScreen자손에 initCursor()와 createGfx()를 실현한다. useOffscreen()를 실현하고 비화면(offscreen)그라픽스기억기를 사용할수 있으면 TRUE를 돌려준다.

⑤ 단지 새로운 QMychipScreen를 돌려주는 작은 함수 qt_get_screen_mychip()를 실현한다.

⑥ qgfxraster_qws.cpp안의 DriverTable표에 자기 구동기를 추가한다. 실례로

{ "MyChip", qt_get_screen_mychip,1 },

첫 파라메터는 자기의 가속구동기를 요구하기 위하여 QWS_DISPLAY와 함께 사용되는 이름이다.

⑦ 새 구동기로 실행하려면

export QWS_DISPLAY=MyChip

(선택적으로 /dev/fb0보다도 각이한 Linux프레임완충기를 요구하려면 MyChip:/dev/fb<n>), 그다음 프로그람을 실행한다

자기 구동기가 PCI나 AGP가 아니면 QLinuxFbScreen대신에 QScreen을 계승해야 하며 QLinuxFbScreen에 류사한 기능을 실현해야 하지만 그렇지 않으면 처리는 비슷해야 한다. 대부분 완성된 실례구동프로그람은 qgfxmach64_qws.cpp이고 qgfxvoodoo_qws.cpp는 더 작고 리해하기 쉬운 구동프로그람을 제공할수 있다.

제5절. Linux틀완충기의 허용

이것은 간단한 차림표이다.

① /usr/src/linux/에 Linux핵심원천코드가 있는가 확인한다.

② 루트로서 가입하고 cd /usr/src/linux를 실행한다.

③ 핵심의 환경을 구성한다.

실행:

make menuconfig

"Code maturity level options"를 선택하고 "Prompt for development and/or incomplete code/drivers"를 설정한다.

그다음 "Console drivers"를 선택하고 "Support for frame buffer devices"를 설정하여 구축한다. 그다음 구동프로그람의 환경을 구성한다. 대부분의 현대그라픽스카드는 "VESA VGA graphics console"을 리용할수 있으며 그것을 사용하거나 자기의

비데오카드와 특별히 일치하는 구동프로그람을 사용한다. 끝으로 "Advanced low level driver options"를 허용하고 16과 32 bpp압축화소유지가 허용된다는것을 확인한다.

완료하려면 exit를 선택하고 보관한다.

④ 핵심을 콤파일한다.

우선

```
make dep
```

다음

```
make bzImage
```

새 핵심은 현재 arch/i386/boot/bzImage에 있어야 한다.

⑤ 핵심을 boot등록부에 복사한다.

```
cp arch/i386/boot/bzImage /boot/linux.vesafb
```

⑥ /etc/lilo.conf를 편집한다.

주의: /etc/lilo.conf의 일시사본을 보관하고 구원디스크를 사용할수 있게 한다. 오유를 범하면 콤퓨터가 기동할수 없다.

파일 /etc/lilo.conf는 체계기동방법을 지정한다. 파일의 정확한 내용은 체계마다 다르다. 여기에 실례가 있다.

```
# LILO configuration file
boot = /dev/hda3
delay = 30
image = /boot/vmlinuz
  root = /dev/hda3
  label = Linux
  read-only # Non-UMSDOS filesystems should be mounted read-only for checking
other=/dev/hda1
    label=nt
    table=/dev/hda
```

처음것의 사본인 새 "image"절을 만든다.

```
image = /boot/linux.vesafb
```

그리고

```
label = Linux-vesafb
```

그것을 바로 첫 화상부분우에 배치한다.

vga = 791을 말하는(1024×768, 16 bpp를 의미하는) 화상부분앞에 한 행을 추가한다.

우의 실례에서 lilo.conf는 현재 다음과 같다.

```
# LILO configuration file
boot = /dev/hda3
delay = 30
vga = 791
image = /boot/linux.vesafb
  root = /dev/hda3
  label = Linux-vesafb
  read-only # Non-UMSDOS filesystems should be mounted read-only for checking
image = /boot/vmlinuz
  root = /dev/hda3
```

label = Linux
read-only # Non-UMSDOS filesystems should be mounted read-only for checking
other=/dev/hda1
label=nt
table=/dev/hda

파일의 현존행들을 변경하지 말고 새 행들을 추가한다.

⑦ 새 변경을 적용하고 lilo프로그람을 실행한다.

lilo

⑧ 체계를 재기동한다. 이제는 체계(혹은 다중처리소자콤퓨터에서 하나이상)가 기동할 때 펭긴새로고를 볼수 있다.

⑨ 새 핵심에서 적당히 기동하지 않으면 LILO재촉문에서 낡은 화상부분의 표식을 입력하여 낡은 핵심으로 기동할수 있다. (실례에서 lilo.conf파일, 낡은 표식은 Linux이다.)

아마 lilo.conf에서 오유로 인하여 이것이 작업하지 않으면 구원디스크로 콤퓨터를 기동하고 일시사본으로부터 /etc/lilo.conf를 되살리고 lilo를 재실행한다.

⑩ 시험: 여기에 틀완충기를 열고 사선으로 채운 적색 바른4각형을 그리는 간단한 프로그람이 있다.

```
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <linux/fb.h>
#include <sys/mman.h>

int main()
{
  int fbfd = 0;
  struct fb_var_screeninfo vinfo;
  struct fb_fix_screeninfo finfo;
  long int screensize = 0;
  char *fbp = 0;
  int x = 0, y = 0;
  long int location = 0;

  // Open the file for reading and writing
  fbfd = open("/dev/fb0", O_RDWR);
  if (!fbfd) {
    printf("Error: cannot open framebuffer device.\n");
    exit(1);
  }
  printf("The framebuffer device was opened successfully.\n");

  // Get fixed screen information
  if (ioctl(fbfd, FBIOGET_FSCREENINFO, &finfo)) {
    printf("Error reading fixed information.\n");
    exit(2);
```

```
	}

	// Get variable screen information
	if (ioctl(fbfd, FBIOGET_VSCREENINFO, &vinfo)) {
	  printf("Error reading variable information.\n");
	  exit(3);
	}

	printf("%dx%d,        %dbpp\n",        vinfo.xres,        vinfo.yres,
vinfo.bits_per_pixel );

	// Figure out the size of the screen in bytes
	screensize = vinfo.xres * vinfo.yres * vinfo.bits_per_pixel / 8;

	// Map the device to memory
	fbp = (char *)mmap(0, screensize, PROT_READ | PROT_WRITE,
MAP_SHARED, fbfd, 0);
	if ((int)fbp == -1) {
	  printf("Error: failed to map framebuffer device to memory.\n");
	  exit(4);
	}
	printf("The   framebuffer   device   was   mapped   to   memory
successfully.\n");

	x = 100; y = 100;     // Where we are going to put the pixel

	// Figure out where in memory to put the pixel
	for ( y = 100; y < 300; y++ )
	  for ( x = 100; x < 300; x++ ) {

	    location = (x+vinfo.xoffset) * (vinfo.bits_per_pixel/8) +
	             (y+vinfo.yoffset) * finfo.line_length;

	    if ( vinfo.bits_per_pixel == 32 ) {
	      *(fbp + location) = 100;    // Some blue
	      *(fbp + location + 1) = 15+(x-100)/2;   // A little green
	      *(fbp + location + 2) = 200-(y-100)/5;  // A lot of red
	      *(fbp + location + 3) = 0;     // No transparency
	    } else  { //assume 16bpp
	      int b = 10;
	      int g = (x-100)/6;    // A little green
	      int r = 31-(y-100)/16;  // A lot of red
	      unsigned short int t = r<<11 | g << 5 | b;
	      *((unsigned short int*)(fbp + location)) = t;
	    }
	  }
```

```
  munmap(fbp, screensize);
  close(fbfd);
  return 0;
}
```

제6절. Qt/Embedded응용프로그람의 실행

Qt/Embedded응용프로그람은 주응용프로그람에서 실행되거나 그 자체가 주응용프로그람이여야 한다. 주응용프로그람은 우선 제일웃준위창문령역과 지적자, 건반입력의 관리를 책임져야 한다.

Qt/Embedded응용프로그람은 QApplication객체를 QApplication::GuiServer형으로 구성하거나 -qws지령행추가선택으로 실행함으로써 주응용프로그람으로 될수 있다.

이 절은 정확히 환경구성된 Linux틀완충기를 가지고있고 어떤 주프로쎄스도 실행하지 않고있는것을 가정한다. 작업중에 있는 Linux틀완충기가 없으면 Qt/Embedded 가상틀완충기를 사용하거나 VNC봉사기로서 Qt/Embedded를 실행할수 있다.

Linux콘솔로 변경하고 실행하려는 실례를 선택한다. 실례로 examples/widgets. $QTDIR가 Qt/Embedded를 설치한 등록부로 설정된것을 확인하고 $QTDIR/lib등록부를 $LD_LIBRARY_PATH에 추가한다. 실례로

export QTDIR=$HOME/qt-VERSION
export LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH

-qws 선택을 지정하여 응용프로그람을 실행한다.

cd $QTDIR/examples/widgets
./widgets -qws

widgets실례가 나타나는가를 알아야 한다. 마우스가 정확히 작업하지 않으면 사용하려는 마우스의 형을 지정해야 한다. 임의의 시각에 Ctrl+Alt+Backspace을 리용하여 마스터응용프로그람을 완료할수 있다.

추가적인 응용프로그람들을 실행하려면 그것들을 의뢰기로서 즉 -qws선택이 없이 실행해야 한다.

1. 현시기

Qt/Embedded는 여러개의 Qt/Embedded주프로쎄스들을 실행하여 여러개의 현시기들을 동시에 사용하게 한다. 이것은 -display지령행파라메터 혹은 $QWS_DISPLAY환경변수를 사용하여 달성된다.

-display파라메터의 문법은 다음과 같다.

  [gfx driver][:driver specific options][:display number]

실례로 fb1에서 mach64구동기를 display 2로서 사용하려면

  $ ./launcher -display Mach64:/dev/fb1:2

이 기능을 실행하려면 다음과 같이 할수 있다.

① VC 1(가상콘솔1)로 변경하고 발사기(launcher)를 실행한다.

  $ cd examples/launcher
  $ ./launcher

② VC 2로 절환하고 다른것을 실행한다.

  $ cd examples/launcher
  $ ./launcher -display :1

다른 발사기가 기동된다. 이 발사기에서 응용프로그람을 기동한다.

③ Ctrl+Alt+F1를 눌러서 display 0으로 돌아온다. 또한 현시기id를 지정하여 특

정한 현시기에서 추가적인 응용프로그람들을 기동할수 있다. VC 3으로 변경한다.

```
$ cd examples/widgets
$ ./widgets -display :1
```

그러면 dislpay :1 (VC 2)에 창문부품실례를 현시한다.

주프로쎄스만이 driver/device부분을 명시적으로 지정할 필요가 있다. 의뢰기들은 련결될 때 주콤퓨터로부터 필요한 정보를 얻는다. 따라서 특별한 구동기를 리용하여 실행하고있는 주봉사기가 있으면 "client -display :n"에 의하여 display n을 사용할수 있다.

2. 마우스입력

Qt/Embedded는 기정으로 마우스를 자동탐지하려고 한다. 유지되여있는 규약은 MouseMan과 Microsoft, IntelliMouse, 기타 일정한 하드웨어에 고유한 장치(실례로 Vr 손접촉조종판)들이다. 사용하려는 마우스를 지정하려면 $QWS_MOUSE_PROTO환경변수를 설정한다. 실례로

```
export QWS_MOUSE_PROTO=IntelliMouse
```

마우스자동탐지는 직렬장치들과 그 장치들을 사용하는 다른 프로그람들과 충돌을 일으키는 psaux를 연다. 그다음 마우스구동기규약과 장치를 명시적으로 지정한다.

제7절. Qt/Embedded에서 문자입력

의뢰기-봉사기통신규약에서는 내부적으로 건누르기와 건놓기는 QWSKeyEvent로서 송신된다. QWSKeyEvent는 표 9-2와 같은 마당들을 포함한다.

표 9-2. QWSKeyEvent의 마당

unicode	유니코드값
keycode	qnamespace.h에서 정의된 Qt건코드값
modifier	Qt::ShiftButton, Qt::ControlButton, Qt::AltButton의 일부로 구성되는 비트마당.
is_press	건누르기이면 TRUE, 건놓기이면 FALSE.
is_auto_repeat	이 사건이 자동반복에 의해 발생하면 TRUE.

봉사기가 건사건을 수신하면 그것을 매개 의뢰기프로쎄스에 송신하고 건사건을 처리하며 정확한 창문에 전송하여 응답한다. 건사건은 여러개의 다른 원천들로부터 온다.

1. 건반구동프로그람

건반구동프로그람은 장치로부터 자료를 읽어서 건사건을 봉사기에 준다.

건반구동프로그람은 서고로 콤파일되거나 플라그인으로서 적재될수 있다. ./configure -help를 실행하여 유효한 건반구동프로그람 들을 렬거한다. tty구동프로그람은 기정환경에서 허용된다.

건반구동프로그람들은 모두 같은 패턴을 따른다. 그것들은 장치로부터 건반자료를 읽어들이고 어느건을 눌렀는가를 알아내고 건반정보를 가지고 정적함수 QWSServer::processKeyEvent()를 호출한다.

지금 콘솔건반구동프로그람은 또한 콘솔절환(Ctrl+Alt-F1...Ctrl+Alt+F10)과 말단(Ctrl+Alt+Backspace)을 조종한다.

새 장치에 대한 건반구동프로그람을 추가하려면 QWSKeyboardHandler와 QKbdDriverPlugin의 파생클라스들을 쓰고 플라그인으로 설치할수 있다.

2. 건사건려파기(입력메쏘드)

봉사기가 건반구동프로그람으로부터 건사건을 수신할 때 그것은 우선 려파기를 통과한다.

이것은 건반에 없는 문자들의 입력을 제공하는 입력방식을 실현하는데 리용될수 있다.

입력방식을 만들려면 QWSServer::KeyboardFilter (src/kernel/qwindowsystem_qws.h에서)의 파생클라스를 만들고 가상함수 filter()를 실현한다. filter()가 FALSE를 돌려주면 사건은 QWSServer::sendKeyEvent()에 의하여 의뢰기들에 전송된다. filter()가 TRUE를 돌려주면 사건은 정지된다. 새 건사건을 생성하려면 QWSServer::sendKeyEvent()을 사용한다. processKeyEvent()를 리용하지 않으므로 무한재귀를 일으킨다.

건반사건려과기를 설치하려면 QWSServer::setKeyboardFilter()를 사용한다. 현재 유일한 려과기는 한번에 설치될수 있다.

려과는 봉사기프로쎄스에서 수행되여야 한다.

launcher실레는 단순한 입력방식의 실례로서 SimpleIM은 파일로부터 환치표를 읽어들인다.

3. 펜입력

건사건은 건반장치로부터 올 필요가 없다. 봉사기프로쎄스는 임의의 시각에 QWSServer::sendKeyEvent()를 호출할수 있다.

일반적으로 이것은 창문부품을 꺼내여 사용자가 점지정장치로 문자들을 지정하게 함으로써 수행된다.

알아두기: 건입력창문부품은 초점을 가지지 말아야 하므로 봉사기는 그때 입력창문부품에로 건사건을 송신하군 한다. 입력창문부품이 절대로 초점을 가지지 않는다는것을 확인하는 한가지 방법은 QWidget구성자에서 WStyle_Customize와 WStyle_Tool창문부품기발들을 설정하는것이다.

Qtopia환경은 Handwriting Recognition과 Virtual Keyboard와 같은 여러가지 입력창문부품들을 포함한다.

제8절. Qt/Embedded의 서체

1. 유지하는 형식들

Qt/Embedded는 4가지 서체형식을 유지한다.

표 9-3. Qt/Embedded의 서체형식

TrueType (TTF)	비레가능서체기술은 현재 MS-Windows와 Apple Macintosh의 표준이며 X11에서 류포되고있다.
Postscript Type1 (PFA/PFB)	인쇄기들에서 흔히 사용하는 비레가능서체, 또한 X11에서도 류포되고있다. 이것들은 TTF서체와 기능이 비슷하다.
Bitmap Distribution Format서체 (BDF)	비레불가능서체의 표준형식. 많은 량의 BDF서체들이 표준X11배포물의 일부로서 제공되며 그 대부분은 Qt/Embedded에서 사용할수 있다. 생산체계에서는 이 서체들을 사용하지 말아야 한다. 이 서체들의 적재는 아주 느리고 많은 기억공간을 차지한다. 그대신 BDF를 QPF로 묘사한다.
Qt Prerendered Font (QPF)	Qt/Embedded에 고유한 가벼운 비레불가능서체형식.

이 서체형식들(늘 허용되는 QPF를 내놓고)의 매개의 유지는 Qt/Embedded특성정

의를 사용함으로써 자유로 허용 혹은 금지할수 있다. 임의의 서체로부터 QPF서체파일을 쓰기 위한 지원이 Qt/Embedded에 있으므로 처음에 TTF와 BDF형식들을 허용하고 필요한 서체와 크기들의 QPF파일들을 보관하고 TTF와 BDF유지에서 삭제한다.

TTF와 BDF로부터 QPF파일들을 생성하는것을 방조하는 도구에 대해서는 tools/makeqpf를 참고하거나 -savefonts선택을 지정하여 자기의 응용프로그람을 실행하시오.

2. 기억기요구

TTF서체에서 주어진 점크기의 서체에서 매개 문자는 그리기나 측도설정조작에서 처음에 사용될 때만 묘사된다. BDF서체들에서 모든 문자는 서체가 사용될 때 묘사된다. QPF서체에서 문자들은 Qt가 그리기에 사용하는것과 같은 형식으로 보관된다.

실례로 아스키문자들을 포함하는 10point Times서체는 QPF형식으로 보관될 때 약 1300byte를 사용한다.

QPF형식이 구조화되는 방식의 우점을 리용하여 Qt/Embedded자료를 읽어서 문법해석하는것보다도 기억기사영한다. 이것은 앞으로 RAM소비를 줄인다.

비례가능서체들은 서체당 더 많은 량의 기억기를 사용하지만 그 매개 서체가 서로 다른 많은 크기를 요구한다면 기억기를 절약하게 한다.

3. 원활한 서체

TTF와 PFA, QPF서체들은 특히 저분해능장치들에서 우수한 읽기능력을 주기 위하여 원활한 비가명식서체들로서 묘사될수 있다. 원활한 서체와 원활하지 않은 서체들사이의 차이는 아래에 설명한다. (차이를 보려면 자기 현시기를 저분해능으로 바꾸어야 한다.)

Unsmooth

그림 9-1 원활하지 않은서체

Smooth

그림 9-2. 원활한서체

4. 유니코드

Qt/Embedded에서 사용하는 모든 서체는 유니코드문자부호화를 사용한다. 오늘 사용할수 있는 대부분의 서체들은 이 부호화를 사용하지만 그것들은 보통 유니코드문자들을 모두 포함하지 않는다. 완전한 16point유니코드서체는 1MB이상의 기억기를 사용한다.

5. 서체정의파일

Qt/Embedded응용프로그람들을 실행할 때 $QTDIR/lib/fonts/fontdir 혹은 /usr/local/qt-embedded/lib/fonts/fontdir라는 파일을 탐색한다. 이 파일은 응용프로그람에 쓸수 있는 서체들을 정의한다. 그것은 다음의 형식을 가진다.

name file renderer italic weight size flags

여기서 파일의 매개 마당의 값들은 표 9-4와 같다.

표 9-4. 파일의 마당값

마당	값
name	Helvetica, Times, 등.
file	helvR0810.bdf, verdana.ttf 등.

마당	값
renderer	BDF 혹은 FT
italic	Y 혹은 n
weight	50은 표준이고, 75은 강조 등.
size	비례가능일 때 0 혹은 점크기 * 10 (즉 12pt일 때 120)
flags	• s: 원활 (비별명식) • u: 보관할 때 유니코드범위(기정은 Latin-1) • a: 보관할 때 ASCII 범위(기정은 Latin-1)

서체정의파일은 QPF서체들을 지정하지 않고 fontdir파일을 포함하는 등록부로부터 직접 적재되며 다음과 같이 이름을 지어야 한다. 즉 *name_size_weightitalicflag*.qpf, 여기서 서체정의파일의 마당값들은 표 9-5와 같다.

표 9-5. 서체정의파일의 마당값

마당	값
name	helvetica, times 등. (소문자에서)
size	point size * 10 (즉 12pt에 대하여 120)
italicflag	경사체에 대하여 i, 그렇지 않으면 아무것도 없다.
weight	50은 표준, 75는 강조 등.

응용프로그람을 -savefonts지령행추가선택으로 실행하면 QPF서체가 아닌 다른 서체를 사용하면 대응하는 QPF파일이 보관된다. 이것은 자기 응용프로그람들의 사용서체를 쉽게 찾고 QPF파일들을 생성하게 하여 Qt/Embedded로부터 TTF와 BDF지원을 금지함으로써 혹은 Qt/Embedded서고원천코드의 kernel/qapplication_qws.cpp에서 qws_savefonts의 초기화를 수정함으로써 자기 응용프로그람들의 기억기사용을 줄일수 있다. 기억기절약의 극단적인 경우에 부분적으로 묘사된 서체(즉 "Product Name™"의 문자들만)들이 서체로부터 요구하는 문자들이라는것이 확실하면 이 서체들을 보관할수 있다. (이 기능에 대해서는 QMemoryManager::savePrerenderedFont()를 참고하시오.)

6. 알아두기

서체정의파일, 서체파일에 대한 명명관례 그리고 QPF파일들의 형식은 3이후의 Qt/Embedded판들에서 다를수 있다.

각이한 회전의 QPF파일들을 생성하려면 프로그람은 QPF출력의 필요한 회전과 일치하는 방향으로 재실행해야 한다. 서체들의 4개 회전 모두를 생성하는 실례는 실제를 완충기에서 다음과 같은것을 실행하는것이다:

```
for dpy in LinuxFb Transformed:Rot90 Transformed:Rot180 Transformed:Rot270
do
  QWS_DISPLAY=$dpy ./makeqpf "$@"
done
```

프로그람들이 한 장치에 대하여 한 방향으로만 실행된다면 하나의 적당한 서체모임이 요구된다.

허용될 때 Qt/Embedded는 강력한 FreeType2서고를 사용하여 TrueType와 Type1지원을 실현한다.

제9절. Qt/Embedded위치지정장치조종

Qt/Embedded에서 위치지정장치조종은 손접촉조종반과 추적구와 같은 마우스나 마우스형 장치에서 작업한다.

보통 오직 하나의 위치지정장치조종만 매몰형장치에 유지되지만 실례를 보여주기 위하여 Qt/Embedded는 많은 량의 지원장치들을 포함한다.

1. 마우스규약

마우스구동기들은 환경스크립트를 거쳐서 허용 또는 금지될수 있다. ./configure -help라고 실행하여 유효한 마우스구동기들을 렬거한다. 기정에서는 pc마우스구동기만이 허용된다.

pc마우스구동기를 허용함으로써 Qt/Embedded는 마우스구동기가 /dev/psaux에서 유지된 형들중 하나이거나 /dev/tty직렬형들중 하나라면 마우스형과 장치를 자동탐지한다. 여러개의 마우스를 탐지하면 모두 동시에 사용할수 있다.

또한 환경변수 QWS_MOUSE_PROTO를 설정하여 사용하려는 마우스를 결정할수 있다. 이 환경변수를 다음과 같이 설정할수 있다.

<protocol>:<device>

여기서 <protocol>는 다음 값들중 하나이다.

- MouseMan
- IntelliMouse
- Microsoft

<device>는 마우스장치 보통 /dev/mouse이다. 그러한 변수를 지정하지 않으면 기정값은 Auto이고 이것은 마우스규약과 장치를 자동탐지하게 한다.

다른 규약을 추가하려면 QWSMouseHandler와 QMouseDriverPlugin의 새로운 파생클라스들을 만들고 플라그인으로 설치할수 있다.

2. 손접촉조종반

Qt/Embedded는 NEC Vr41XX손접촉조종판과 iPAQ와 Zaurus에서 사용되는 로출식linux손접촉화면기능을 적재한다. 이것들은 QWSCalibratedMouseHandler의 파생클라스이고 QWSCalibratedMouseHandler는 QWSMouseHandler의 파생클라스로서 embedded/qmouse_qws.cpp에 있다.

제10절. Qt/Embedded환경변수

표 9-6. Qt/Embedded환경변수

변 수	설 명
QWS_SW_CURSOR	하드웨어유표를 유지하는 가속구동프로그람을 사용할 때에도 쏘프트웨어마우스유표를 사용한다.
QWS_DISPLAY	현시기형과 틀완충기를 정의한다. 실례로 Voodoo3 Mach64:/dev/fb1 /dev/fb0.에 대하여 비가속화된 Linux틀완충기구동프로그람에 대한 기정값이다. 유효구동프로그람들은 QVfb, VGA16, LinuxFb(비가속Linux틀완충기), Mach64(Rage Pro와 같은 ATI Mach64카드용으로 가속화됨), Voodoo3(3dfx Voodoo 3용으로 가속화됨. 또한 Voodoo Banshee에서 작업해

	야 한다.), Matrox (Matrox Millennium의 Matrox그라픽스카드 모두에서 작업해야 한다.), Transformed (회전현시기용), SVGALIB 그리고 VNC이다. 변형된 현시기는 특수한 형식을 가진다. Rot<x>로서 90도의 배수로 회전을 지정해야 한다. 실례로 Transformed:Rot90을 들수 있다.
QTDIR	Qt/Embedded에게 그 서체들을 찾을 위치를 알려준다. fontdir는 $QTDIR/etc/fonts/에 있어야 한다. 정의되지 않으면 /usr/local/qt-embedded 로 가정한다.
QWS_SIZE	Qt/Embedded는 화면중심에 <폭>×<높이>크기의 창문으로 된다. 실례로 320×200.
QWS_NOMTRR	x86상에서 써넣기결합된것으로서 틀완충기를 정의할 때 Memory Type Range Register들을 사용하지 않는다. 써넣기결합은 그라픽스출력을 가속화한다.
QWS_CARD_SLOT	가속화된 구동프로그람들에 어느 카드를 가속화하려고 하는가를 알려준다. 이것은 /proc/bus/pci경로안에 있어야 한다. 기정은 /proc/bus/pci/01/00.0인데 보통 AGP카드체계에서 둘째PCI모선우의 첫째 장치이다.
QWS_USB_KEYBOARD	/dev/tty를 열 대신에 QWS_USB_KEYBOARD에서 정의된 USB저준위사건장치(실례로 /dev/input/event0)를 연다. 보통 각이한 틀완충기들상에서 X와 Qt/Embedded를 나란히 실행하려는 경우에 이렇게 한다.
QWS_MOUSE_PROTO	<type>:<device>로서 정의한다. 실례로 Microsoft:/dev/ttyS0이다. USB마우스를 직접(X와는 별개로) 사용하려고 한다면 MouseMan:/dev/input/mouse0 혹은 그와 류사한것을 사용한다. 유효한 마우스규약들로서 Auto(자동수감통신규약), MouseMan, IntelliMouse, Microsoft, QVfbMouse (QVfb에서만 유용) 그리고 간단한 견본손접촉조종반구동프로그람 TPanel이 있다.
QWS_KEYBOARD	건반형을 정의한다. 한번에 여러개의 건반을 조종할수 있으며 입력은 그 모두로부터 읽어들일수 있다. 유효값들은 Buttons (QT_QWS_IPAQ가 콤파일되면 iPaq단추장치, 그렇지 않으면 Cassiopeia용의것), QVfbKeyboard (QVfb에서만 유용), 그리고 TTY (QWS_USB_KEYBOARD가 정의되는가에 따라 USB 건반이나 /dev/tty)

제11절. 자기의 응용프로그람을 Qt/Embedded에 이식

가동환경에 의존하는 코드가 없는 현존Qt응용프로그람들은 이식을 제공할것을 요구하지 말아야 한다. 가동환경에 의존하는 코드는 기초하고있는 창문체계(Windows 혹은 X11)에로의 호출인 체계호출과 QApplication::x11EventFilter()와 같은 Qt가동환경에 고유한 메쏘드를 포함한다.

가동환경에 고유한 코드를 사용할 필요가 있는 경우에 #ifdef 지령에 의하여 각 가동환경에 대하여 코드를 허용 및 금지하는데 쓰일수 있는 정의된 마크로들이 있다.

표 9-7. 이식과 관련한 정의된 마크로

가동환경	마크로
Qt/X11	Q_WS_X11
Qt/Windows	Q_WS_WIN
Qt/Embedded	Q_WS_QWS

또한 Qt/Embedded는 응용프로그람들을 콤파일할 때 다음의 기발들을 정의할것을 요구한다.

-DQWS -fno-exceptions -fno-rtti

례외와 RTTI는 크기와 속도에서 큰 추가비용을 요구하므로 Qt/Embedded에서 허용되지 않는다.

제12절. Qt/Embedded를 이식할 때 알아야 할 문제

Qt/Embedded는 가동환경에 의존하지 않도록 설계된다. 오직 공개적으로 사용할수 있는 판은 Linux실현이다. 다음과 같은 의존관계는 다른 조작체계에 이식하려는 경우에 설명할 필요가 있다.

• System V IPC(공유기억기와 씨마퍼)는 의뢰기와 봉사기사이에 창문령역들을 공유하는데 리용된다. 단일응용프로그람설치(즉 봉사기인 오직 하나의 프로그람의 실행)를 요구하지 않으면 류사한것을 제공할 필요가 있다. 또한 System V씨마퍼는 틀완충기의 동기호출에 사용된다.

qwindowsystem_qws.cpp, qwsregionmanager_qws.cpp, qapplication_qws.cpp, 그리고 qlock_qws.cpp를 수정한다.

• **Unix령역소케트**는 응용프로그람들사이에서 건반사건들과 창문, QCOP통보문들을 발생시킬데 대한 요구를 전송하는데 사용된다. 또한 단일응용프로그람설치를 요구하지 않으면 류사한것을 제공할 필요가 있다. 통보문대기렬이나 그와 류사한 기구들을 리용하여 이와 같은것을 실현할수 있어야 하며 QCOP통보문(의뢰기응용프로그람과 Qt/Embedded아닌 응용프로그람들에 의하여 생성되는것)을 제외하고 개별적 통보문들은 여러 바이트의 길이만 가져야 한다.

qwssocket_qws.cpp를 수정한다.

• **Linux틀완충장치**는 그리기령역에서 변환하는데 쓰인다. 기억기변환된 틀완충기에로의 바이트지적자와 폭과 높이 그리고 비트깊이에 대한 정보(간단히 고정코드작성할수 있는것)를 주는 다른것들로 (QScreen의 새 클라스를 창조하여) 그것을 교체할 필요가 있다. 자기의 틀완충기가 기억기변환되지 않거나 유지안된 형식이나 깊이가 아니면 QGfxRaster도 수정해야 한다.

qgfxlinuxfb_qws.cpp를 수정한다.

• **가속구동프로그람**들은 현재 Linux QScreen을 리용하며 /proc/bus/pci를 사용하여 PCI환경공간에서 사영한다. 그러나 이것들은 오직 실례구동프로그람들뿐이고 임의

의 경우에 아마 자체의 구동프로그람을 쓸 필요가 있으며 조종등록기들에서 자체의 사영방법을 제공해야 한다.

qgfxmach64_qws.cpp, qgfxvoodoo_qws.cpp 그리고 qgfxmatrox_qws.cpp를 수정한다.

• **음성**은 Linux /dev/dsp형식의 장치를 사용한다. Qt/Embedded음성봉사기를 사용하려고 한다면 그것을 재정의해야 한다.

qsoundqss_qws.cpp를 수정한다.

• select()는 QSocketDevice들을 실현하고 Qt/Embedded봉사기응용프로그람들에 오고가는 사건들을 접수하는데 쓰인다.

qapplication_qws.cpp를 수정한다.

Qt/Embedded는 표준C서고와 일부 Posix함수들을 사용하게 한다. 주로 Posix함수들은 가동환경에 의존하는 코드(실례로 Linux틀완충기에서 사영하는데 mmap())에 집중된다.

제13절. Qt/Embedded성능조정

저전원장치들에서 매몰형응용프로그람들을 구축할 때 탁상응용프로그람환경에서 고려할수 없는 많은 선택을 사용할수 있다. 이 선택들은 다음 인자들을 희생시키여 기억기와 CPU요구를 줄인다.

1. 일반적인 프로그람작성형식

대화칸와 창문부품들을 필요할 때마다 창조하고 삭제하지 않고 일단 창조한 다음 QWidget::hide() 및 QWidget::show()를 호출하여 은페하거나 표시한다. 이것은 기억기를 좀 더 많이 사용하지만 훨씬 더 고속이다.

2. 정적련결 대 동적련결

많은 CPU와 기억기는 ELF결합처리에 의하여 리용된다. 한조의 응용프로그람을 정적구축하여 기억기를 절약할수 있다. 이것은 동적서고(libqte.so)와 그 서고에 동적으로 련결하는 실행파일들의 집합을 가지는것보다 모든 응용프로그람들을 하나의 실행파일로 구축하고 그것을 정적서고(libqt.a)에 정적으로 련결한다는것을 의미한다. 이것은 기동시간을 줄이고 기억사용량을 줄이지만 유연성을 떨구고(새 응용프로그람을 추가하려면 하나의 실행파일을 재콤파일해야 한다.) 튼튼하지 못하다(하나의 응용프로그람에 오유가 있으면 다른 응용프로그람들을 상하게 할수 있다). 말단사용자응용프로그람을 설치해야 한다면 이것은 선택하지 않을수 있으나 제한된 CPU능력과 기억기를 가지는 장치용으로 하나의 응용프로그람을 구축하는 경우에는 이 선택이 아주 유리하다.

정적서고로서 Qt를 콤파일하려면 configure를 실행할 때 -static선택을 추가한다.

응용프로그람조를 모두 하나의 응용프로그람으로 구축하려면 매개 응용프로그람을 main()함수에 최소의 코드만 가지는 독자의 창문부품이나 창문부품모임으로 설계한다. 그다음 응용프로그람들사이를 절환하는 방법(실례로 QIconView)을 주는 응용프로그람을 쓴다. Qtopia가 그 실례이다. 그것은 일련의 동적련결실행파일들로서 혹은 하나의 정적응용프로그람으로서 구축될수 있다.

일반적으로 아직 표준C서고와 자기 장치에서 다른 응용프로그람들이 사용할수 있는 다른 서고들에 대해서만 동적으로 련결해야 한다.

3. 다른 기억할당

일부 가동환경들에서 C++콤파일러들이 적재하는 서고들은 new와 delete연산자들의 기능이 빈약하므로 이 함수들을 재정의하여 성능을 높일수 있다. 실례로 자기 코드에

다음과 같이 추가하여 보통의 C할당자들로 절환할수 있다.

```
void* operator new[]( size_t size )
{
  return malloc( size );
}
void* operator new( size_t size )
{
  return malloc( size );
}
void operator delete[]( void *p )
{
  free( p );
}
void operator delete[]( void *p, size_t size )
{
  free( p );
}
void operator delete( void *p )
{
  free( p );
}
void operator delete( void *p, size_t size )
{
  free( p );
}
```

제14절. VNC봉사기로서의 Qt/Embedded

VNC규약은 망의 어디서나 콤퓨터의 현시기에 표시하고 대화할수 있게 한다.

이러한 방법으로 Qt/Embedded를 사용하려면 -qt-gfx-vnc선택을 지정하여 Qt의 환경을 구성하고 16bit현시기유지가 가능하다는것을 담보한다. 다음과 같이 자기의 응용프로그람을 실행한다.

```
application -display VNC:0
```

그다음 자기 응용프로그람을 실행하고있는 콤퓨터에서 VNC의뢰기위치지정을 실행한다. 실례로 X11 VNC의뢰기를 리용하여 같은 콤퓨터로부터 응용프로그람을 표시한다.

```
vncviewer localhost:0
```

기정으로 Qt/Embedded는 640×480화소의 현시기를 창조하다. QWS_SIZE환경변수를 다른 크기 실례로 QWS_SIZE=240×320으로 설정하여 이것을 변경할수 있다.

VNC의뢰기들은 현시기체계(즉 X11과 Windows, Amiga, DOS, VMS, 기타)들의 거대한 배렬에 사용할수 있다.

Qt가상틀완충기는 다른 기술이다. 그것은 공유기억기를 사용하므로 훨씬 더 빠르고 원활하지만 망에서는 조작하지 않는다.

부록 1. Qt클라스서고

1. Qt의 기본클라스들

제일 흔히 사용하는 Qt클라스들은 다음과 같다.

QAction	QFileDialog	QMessageBox	QSlider	QTimer
QApplication	QFont	QMovie	QSound	QToolBar
QButtonGroup	QFontDialog	QNetworkProtocol	QSpinBox	QToolBox
QCanvas	QGLWidget	QObject	QSplashScree n	QToolButton
QCheckBox	QGridLayout	QPainter	QSplitter	QToolTip
QClipboard	QGroupBox	QPalette	QSql	QTranslator
QColorDialog	QGuardedPtr	QPen	QSqlDatabase	QUrl
QComboBox	QHBoxLayout	QPixmap	QSqlDriverPl ugin	QUrlOperator
QDataBrowser	QIconSet	QPoint	QSqlForm	QValidator
QDataTable	QIconView	QPopupMenu	QSqlQuery	QValueList
QDataView	QImage	QPrinter	QStatusBar	QValueStack
QDate	QInputDialog	QProcess	QString	QValueVector
QDateEdit	QLabel	QProgressBar	QStringList	QVariant
QDateTime	QLCDNumber	QProgressDialog	QTabDialog	QVBoxLayout
QDateTimeEdit	QLibrary	QPushButton	QTable	QWhatsThis
QDial	QLineEdit	QRadioButton	QTabWidget	QWidget
QDialog	QListBox	QRect	QTextBrowse r	QWidgetStack
QDict	QListView	QRegExp	QTextEdit	QWizard
QDir	QMainWindow	QScrollView	QTextStream	QWorkspace
QDockWindow	QMap	QSettings	QTime	QXmlSimpleReade r
QFile	QMenuBar	QSimpleRichText	QTimeEdit	

2. 클라스의 분류

여기서는 Qt의 클라스들을 분류한다. 일부 클라스들은 하나이상의 부류에 속할수 있다.

부류	설명
추상창문부품	파생클라스작성을 통하여 사용할수 있는 추상창문부품클라스들.
고급한 창문부품	목록보기와 진척상황띠와 같은 고급한 GUI 창문부품.
기본창문부품	단추, 복합칸, 흐름띠와 같은 기본GUI 창문부품.
자료기지	자료기지관련클라스들, 실례로 SQL자료기지.
날자와 시간	날자와 시간을 처리하는 클라스들.
끌기와 놓기	끌어다놓기와 mime형부호화와 복호화를 취급하는 클라스들.
환경	사건처리, 체계설정호출, 국제화와 같은 여러가지 대역봉사를 제공하는 클라스들.
사건	사건을 창조하고 처리하는데 쓰이는 클라스들.
편의클라스	list, queue, stack, string과 같은 집합클라스들과 QApplication을 요구하지 않고 쓰일수 있는 클라스들.

부류	설명
그라픽스와 인쇄	OpenGL을 비롯하여 그리기(와 인쇄)기초를 제공하는 클라스들. (또한 화상처리와 다매체를 참고)
방조체계	응용프로그람의 직결방조를 제공하는데 쓰이는 클라스들.
화상처리	수자화상 부호화와 복호화, 조작. (또한 그라픽스와 인쇄 및 다매체를 참고)
배치관리	복잡한 대화칸을 만들기 위하여 창문부품들의 자동크기조절과 이동을 처리하는 클라스들.
암시적 및 명시적 공유클라스	고속복사에 참고계수를 사용하는 클라스들.
입출력과 망구축	파일입출력과 등록부와 망조종을 제공하는 클라스들.
MainWindow 및 관련클라스	차림표, 도구띠, 작업공간 등 전형적이고 현대적인 기본응용프로그람창문에 필요한 모든것.
기타	여러가지 기타 유용한 클라스들
다매체	그라픽스, 음성, 동화상 등에 대한 유지를 제공하는 클라스들. (또한 그라픽스와 인쇄 및 화상처리를 참고)
객체모형	객체모형에 기초하는 Qt GUI개발도구.
조직자	분할기, 타브띠, 단추그룹 등 사용자대면부조직자들.
플라그인클라스	플라그인관련클라스들.
표준대화칸	파일, 본트, 색선택 등을 위한 준비된 클라스들.
형판서고	Qt의 형판서고용기클라스들.
본문관련클라스	본문처리용 클라스들. (또한 XML클라스들을 참고.)
스레드클라스	스레드유지를 제공하는 클라스들.
창문부품외관	형식, 서체, 색 등의 외관전용화.
XML클라스	실례로 DOM과 SAX를 통하여 XML을 지원하는 클라스들,.

1) 추상적인 창문부품클라스

이 클라스들은 추상창문부품이므로 보통 자체로 쓰이지 못하지만 계승용으로 사용할수 있는 기능들을 제공한다.

QButton	단추에 대한 일반적인 기능을 제공하는 단추창문부품의 추상기초클라스
QCanvas	QCanvasItem객체들을 포함할수 있는 2D령역
QDialog	대화창문의 기초클라스
QFrame	틀을 소유할수 있는 창문부품들의 기초클라스
QGridView	크기가 고정된 격자표의 추상기초클라스
QScrollView	조종가능한 흐름띠를 가지는 흘림령역
QWidget	모든 사용자대면부객체용 기초클라스
QWizard	위자드대화칸용 틀거리

2) 고급한 창문부품들

다음의 클라스들은 더 복잡한 사용자대면부조작을 제공한다.

QCheckListItem	검사가능한 목록보기항목
QCheckTableItem	QTable의 검사칸
QComboTableItem	QTable에서 복합칸을 사용하는 수단
QDateEdit	날자편집기
QDateTimeEdit	QDateEdit과 QTimeEdit창문부품을 날자시간편집용 단일창문부품에 결합한다.
QDesktopWidget	다중head체계에서 화면정보에 대한 호출
QHeader	표나 목록보기용 머리부 행 또는 렬
QIconFactory	QIconSet용 픽스매프를 창조하는데 사용
QIconView	이동가능한 표식그림기호를 가진 령역
QIconViewItem	QIconView안의 한개 항목
QListBox	선택가능한 읽기전용의 항목목록
QListBoxPixmap	픽스매프와 선택본문을 가진 목록칸항목
QListBoxText	본문을 현시하는 목록칸항목들
QListView	목록/나무보기를 실현한다.
QListViewItem	목록보기항목을 실현한다.
QListViewItemIterator	QListViewItem들의 집합용 반복자
QMultiLineEdit	본문입력용 단순편집기
QProgressBar	수평진행상황띠
QTab	QTabBar의 구조체들
QTabBar	타브띠 실례로 타브형식대화칸에서 사용
QTable	유연성있는 편집가능표창문부품
QTableItem	QTable용 세포항목
QTableSelection	QTable의 선택령역에로의 호출
QTabWidget	타브형식창문부품들의 탄창
QTextBrowser	초본문련결을 가진 리치본문열람기
QTimeEdit	시간편집기
QToolBox	타브형식창문부품항목들의 렬

3) 기초창문부품들

기본조종요소(창문부품)들은 직접 사용하는것으로 설계된다. 또한 파생클라스를 만들기 위한 추상창문부품클라스들과 그밖에 더 복잡한 창문부품들도 있다.

QAction	차림표와 도구띠에 나타나는 추상적인 사용자대면부작용.
QActionGroup	작용들을 한 그룹에 묶는다.
QCheckBox	본문표식을 가진 검사칸
QComboBox	본문표식을 가진 복합칸
QDial	(속도계나 전위차계와 같은) 원형의 범위조종
QLabel	본문 혹은 화상현시
QLCDNumber	LCD형식으로 수를 현시한다.
QLineEdit	한행본문편집기
QPopupMenu	올리펼침차림표창문부품
QPushButton	지령단추
QRadioButton	본문이나 픽스매프표식을 제공하는 라지오단추
QScrollBar	수직 혹은 수평흐름띠

QSizeGrip	제일웃준위창문용 구석격자
QSlider	수직 혹은 수평슬라이더
QSpinBox	스핀칸창문부품(스핀단추)
QSyntaxHighlighter	QTextEdit문법강조표시기를 실현하는 기초클라스
QTextEdit	강력한 단일페지리치본문편집기
QToolButton	QToolBar안에서 보통 사용되는 지령이나 선택들의 고속호출단추

4) 자료기지클라스

다음의 클라스들은 SQL자료기지의 호출을 제공한다.

QDataBrowser	자료항목폼에서 자료조작과 항행
QDataTable	열람 및 편집을 유지하는 유연성있는 SQL표
QDataView	읽기전용SQL폼
QEditorFactory	QVariant자료형용 편집칸창문부품창조에 쓰인다.
QSql	대역일것을 요구하는 Qt SQL식별자용 이름공간
QSqlCursor	SQL표와 보기의 열람과 편집
QSqlDatabase	SQL자료기지련결을 창조하고 일괄처리를 제공하는데 리용
QSqlDriver	SQL자료기지호출용 추상기초클라스
QSqlEditorFactory	QDataTable과 QSqlForm에서 리용하는 편집기를 창조하는데 리용
QSqlError	SQL자료기지오유정보
QSqlField	SQL자료기지표와 보기들에서 마당을 조작한다.
QSqlFieldInfo	SQL마당과 련상된 메타자료를 보관한다.
QSqlForm	SQL자료기지에 련결된 자료항목폼을 창조하고 관리한다.
QSqlIndex	QSqlCursor와 QSqlDatabase색인을 조작 및 서술하는 기능
QSqlPropertyMap	창문부품을 SQL마당으로 변환하는데 리용
QSqlQuery	SQL명령문의 실행 및 조작수단
QSqlRecord	자료기지레코드 즉 자료기지마당들의 모임을 은페한다.
QSqlRecordInfo	자료기지마당메타자료의 모임을 은페한다.
QSqlResult	SQL자료기지로부터 자료를 호출하기 위한 추상대면부
QSqlSelectCursor	일반SQL SELECT문의 열람

5) 시간과 날자

이 클라스들은 체계에 의존하지 않는 날자와 시간추상화를 제공한다.

또한 다음것을 사용할수 있다. Qt2000년호환명령문.

QDate	날자함수
QDateEdit	날자편집기
QDateTime	날자와 시간함수
QDateTimeEdit	QDateEdit와 QTimeEdit창문부품을 날자시간을 편집하는 하나의 창문부품으로 결합한다
QTime	시간함수
QTimeEdit	시간편집기

QTimer	시계신호와 단일발사시계

6) 끌기와 놓기 클라스

다음의 클라스들은 끌어다놓기와 필요한 mime형 부호화와 복호화를 론의한다.

QColorDrag	색들을 전송하기 위한 끌어다놓기객체
QDragEnterEvent	끌어다놓기가 처음으로 창문부품우로 끌기할 때 그 창문부품에 송신되는 사건
QDragLeaveEvent	끌어다놓기가 창문부품을 떠날 때 그 창문부품에 송신되는 사건
QDragMoveEvent	끌어다놓기가 진행중일 때 송신되는 사건
QDragObject	MIME에 기초한 자료전송을 은폐한다.
QDropEvent	끌어다놓기가 끝났을 때 송신되는 사건
QIconDrag	QIconView안에서 끌어다놓기조작을 유지한다.
QIconDragItem	끌기항목을 은폐한다.
QImageDrag	화상을 전송하기 위한 끌어다놓기객체
QMacMime	열린표준형MIME를 Mac형식으로 변환한다.
QMimeSource	일정한 MIME형의 형식화된 자료를 제공하는 객체들의 추상화
QStoredDrag	임의의 MIME자료를 위한 간단히 기억값끌기객체
QTextDrag	일반본문과 유니코드본문을 변환하기 위한 끌어다놓기객체
QUriDrag	URI참고목록용 끌기객체
QWindowsMime	공개표준MIME를 Window오려둠판형식으로 변환한다.

7) 환경클라스

다음의 클라스들은 사건조종, 체계설정, 국제화와 같은 여러가지 대역봉사를 응용프로그람에 제공한다.

QClipboard	창문체계오려둠판에 대한 호출
QDesktopWidget	다중헤드체계상에서 화면정보호출
QEvent	모든 사건클라스들의 기초클라스. 사건객체는 사건파라메터들을 포함한다.
QFontDatabase	기본창문체계에서 사용할수 있는 서체에 대한 정보
QMimeSourceFactory	MIME형자료의 확장가능한 제공기
QMutex	스레드들사이의 호출계렬화
QMutexLocker	QMutexe의 잠그기와 해제를 간단화한다.
QPixmapCache	픽스매프용의 응용프로그람대역캐쉬
QSemaphore	건전한 완전쎄마훠
QSessionManager	쎄숀관리기에 대한 호출
QThread	가동환경에 의존하지 않는 스레드
QThreadStorage	스레드별 자료기억
QTranslator	본문출력을 위한 국제화지원
QTranslatorMessage	번역기통보문과 그 속성들
QWaitCondition	스레드들사이의 조건들에 대한 대기/동작(waking)을 허용한다.

8) 사건클라스

다음의 클라스들은 사건의 창조와 조종에 리용된다.

QChildEvent	자식객체사건들을 위한 사건파라메터
QCloseEvent	닫기사건을 서술하는 파라메터
QContextMenuEvent	상황차림표사건을 서술하는 파라메터들
QCustomEvent	사용자정의사건지원
QDragEnterEvent	끌어다놓기가 처음으로 창문부품우로 끌기할 때 그 창문부품에 송신되는 사건
QDragLeaveEvent	끌어다놓기가 창문부품을 떠날 때 그 창문부품에 송신되는 사건
QDragMoveEvent	끌어다놓기가 진행중일 때 송신되는 사건
QDropEvent	끌어다놓기가 끝났을 때 송신되는 사건
QEvent	모든 사건클라스들의 기초클라스. 사건객체는 사건파라메터들을 포함한다.
QEventLoop	사건대기렬을 관리한다.
QFocusEvent	창문부품초점사건의 사건파라메터
QHideEvent	창문부품이 은페된 후 전송되는 사건
QIconDragEvent	기본그림기호끌기를 시작한 신호
QIMEvent	입력메쏘드사건용 파라메터들
QKeyEvent	건사건을 서술한다.
QMouseEvent	마우스사건을 서술하는 파라메터
QMoveEvent	이동사건을 위한 사건파라메터
QPaintEvent	그리기사건을 위한 사건파라메터
QResizeEvent	크기조절사건용 사건파라메터
QShowEvent	창문부품이 표시될 때 송신되는 사건
QTabletEvent	Tablet사건을 서술하는 파라메터
QTimer	시계신호와 단일소리시계
QTimerEvent	시계사건을 서술하는 파라메터
QWheelEvent	바퀴사건을 서술하는 파라메터들

9) 비GUI클라스들

비GUI클라스들은 GUI클라스들과는 독립적으로 사용할수 있는 일반목적집합과 문자렬클라스들이다.

특히 이 클라스들은 QApplication에 전혀 의존되지 않으므로 비GUI프로그람들에서 사용할수 있다.

QAsciiCache	char*건에 기초한 캐쉬를 제공하는 형판클라스
QAsciiCacheIterator	QAsciiCache집합용 반복자
QAsciiDict	char*건에 기초한 사전을 제공하는 형판클라스
QAsciiDictIterator	QAsciiDict집합용 반복자
QBitArray	비트배렬
QByteArray	바이트배렬
QCache	QString건에 기초한 캐쉬를 제공하는 형판클라스
QCacheIterator	QCache집합용 반복자
QCString	전형적인 C의 0으로 완료되는 char배렬(char *)의 추상화

QDeepCopy	암시적공유와 명시적공유클라스들이 유일한 자료를 참고한다는것을 담보하는 형판클라스
QDict	QString건에 기초한 사전을 제공하는 형판클라스
QDictIterator	QDict집합의 반복자
QIntCache	긴열쇠에 기초하는 캐쉬를 제공하는 형판클라스
QIntCacheIterator	QIntCache집합의 반복자
QIntDict	긴열쇠에 기초하는 등록부를 제공하는 형판클라스
QIntDictIterator	QIntDict집합의 반복자
QMap	사전을 제공하는 값에 기초한 형판클라스
QMapConstIterator	QMap의 상수반복자
QMapIterator	QMap의 반복자
QMemArray	단순형배렬을 제공하는 형판클라스
QObjectList	QObject들의 QPtrList
QObjectListIterator	QObjectList용 반복자
QPair	한쌍의 원소들을 제공하는 값기초형판클라스
QPtrCollection	대다수 지적자에 기초한 Qt집합의 기초클라스
QPtrDict	void*건에 기초하는 사전을 제공하는 형판클라스
QPtrDictIterator	QPtrDict집합용 반복자
QPtrList	목록을 제공하는 형판클라스
QPtrListIterator	QPtrList집합용 반복자
QPtrQueue	대기렬을 제공하는 형판클라스
QPtrStack	탄창을 제공하는 형판클라스
QPtrVector	벡토르(배렬)을 제공하는 형판집합
QRegExp	정규식들을 사용한 패턴대조
QStrIList	대소문자식별비교기능을 가진 char*의 2중련결목록
QString	유니코드본문과 전형적인 C의 '\0'완료문자배렬의 추상화
QStringList	문자렬목록
QStrList	char*의 2중련결목록
QStrListIterator	QStrList와 QStrIList클라스들의 반복자
QValueList	목록을 제공하는 값기초형판클라스
QValueListConstIterator	QValueList의 상수반복자
QValueListIterator	QValueList의 반복자
QValueStack	탄창을 제공하는 값기초형판클라스
QValueVector	동적배렬을 제공하는 값기초형판클라스

10) 그라픽스클라스들

다음의 클라스들은 2D와 (OpenGL에 의한) 3D용의 강력한 그라픽스그리기를 제공한다.

QBitmap	흑백픽스매프(1 bit 깊이)
QBrush	QPainter로 그려질 도형들의 도색패턴을 정의한다.
QCanvas	QCanvasItem객체를 포함할수 있는 2D령역
QCanvasEllipse	QCanvas상의 타원 혹은 타원토막

QCanvasItem	QCanvas 상의 추상도형객체
QCanvasItemList	QCanvasItem들의 목록
QCanvasLine	QCanvas상의 직선
QCanvasPixmap	QCanvasSprite용 픽스매프
QCanvasPixmapArray	QCanvasPixmap배렬
QCanvasPolygon	QCanvas상의 다각형
QCanvasPolygonalItem	QCanvas상의 다각형그림항목
QCanvasRectangle	QCanvas상의 직4각형
QCanvasSpline	QCanvas상의 다중베쎌spline
QCanvasSprite	QCanvas상의 동화그림항목
QCanvasText	QCanvas상의 본문객체
QCanvasView	QCanvas의 화면보기
QColor	RGB 혹은 HSV값에 기초한 색
QColorDialog	색을 지정하기 위한 대화칸창문부품
QColorGroup	창문부품색의 그룹
QDirectPainter	비데오하드웨어의 직접호출
QFont	본문그리기에 사용하는 서체
QFontDatabase	기본창문체계에서 사용할수 있는 서체에 대한 정보
QFontInfo	서체에 대한 일반적 정보
QFontMetrics	서체크기정보
QGL	Qt OpenGL모듈에서 여러가지 식별자를 위한 이름공간
QGLColormap	QGLWidget들에 사용자정의색략도를 설치하는데 쓰인다.
QGLContext	OpenGL묘사상황을 은페한다.
QGLFormat	OpenGL묘사상황의 현시형식
QGLWidget	OpenGL그라픽스묘사용 창문부품
QIconSet	여러가지 형식과 크기의 그림기호모임
QImage	화소자료에 대한 직접호출을 가진 하드웨어독립픽스매프표시
QImageConsumer	QImageDecoder에 의하여 사용되는 추상화
QImageDecoder	유지되여있는 모든 화상형식의 증분식화상해신기
QImageFormat	특수화상형식의 증분식화상해신기
QImageFormatType	QImageFormat객체를 만드는 공장
QImageIO	화상의 적재와 보관용 파라메터
QMovie	동화상이나 정지화상의 적재상황을 신호하는 증분식적재
QPaintDevice	그리기할수 있는 객체들의 기초클라스
QPaintDeviceMetrics	그리기장치에 대한 정보
QPainter	창문부품들에 대한 저수준그리기를 한다.
QPalette	매개 상태를 위한 색그룹
QPen	QPainter가 직선과 륜곽선을 그리는 방법을 정의한다.
QPicture	QPainter지령을 기록하고 재연시하는 그리기장치
QPixmap	비직결화면, 화소에 기초한 그리기장치
QPixmapCache	픽스매프용의 응용프로그람대역캐쉬
QPNGImagePacker	잘 압축된 PNG동화상을 창조한다.
QPoint	평면안의 점을 정의한다.

QPointArray	점들의 배렬
QPrinter	인쇄기에 그리는 그리기장치
QRect	평면안의 직사각형을 정의한다.
QRegion	그리기프로그람용의 복사칸(clip)령역
QSize	2차원객체의 크기를 정의한다.
QStyleSheet	리치본문묘사용의 형식들의 집합과 타그생성기
QWMatrix	자리표계의 2D변환

11) 방조체계

다음의 클라스들은 자기의 응용프로그람에서 직결방조의 모든 형식을 3준위로 자세히 제공한다.

1. 도구암시와 상태띠통보는 매우 가볍고 극히 간단하며 사용자대면부에서 완전히 통합되여있으며 호출하기 위한 사용자교제가 적거나 없을것을 요구한다.

2. What's This?는 매우 간단한 설명이다.

3. 직결방조는 임의의 량의 정보를 포함할수 있으나 호출속도가 뜨고 사용자의 작업과 분리되여있다.

QStatusBar	상태정보를 표시하는데 적합한 수평띠
QStyleSheet	리치본문묘사용의 형식들의 집합과 타그생성기
QTextBrowser	초본문항행이 가능한 리치본문열람기
QToolTip	임의의 창문부품을 위한 도구암시(풍선형방조) 혹은 창문부품의 직4각형부분
QToolTipGroup	도구암시들을 관련된 그룹들로 집합한다.
QWhatsThis	《이것은 무엇인가?》와 같은 질문에 대답하는 임의의 창문부품의 간단한 서술

12) 화상처리클라스

이 클라스들은 화상조작에 사용된다.

QBitmap	흑백픽스매프(1 bit 깊이)
QBrush	QPainter로 그려질 도형들의 도색패턴을 정의한다.
QCanvas	QCanvasItem객체를 포함할수 있는 2D령역
QCanvasEllipse	QCanvas상의 타원 혹은 타원토막
QCanvasItem	QCanvas 상의 추상도형객체
QCanvasItemList	QCanvasItem들의 목록
QCanvasLine	QCanvas상의 직선
QCanvasPixmap	QCanvasSprite용 픽스매프
QCanvasPixmapArray	QCanvasPixmap배렬
QCanvasPolygon	QCanvas상의 다각형
QCanvasPolygonalItem	QCanvas상의 다각형그림항목
QCanvasRectangle	QCanvas상의 직4각형
QCanvasSpline	QCanvas상의 다중베쩰 스플라인
QCanvasSprite	QCanvas상의 동화그림천항목
QCanvasText	QCanvas상의 본문객체
QCanvasView	QCanvas의 화면보기
QColor	RGB 혹은 HSV값에 기초한 색

QColorGroup	창문부품색의 그룹
QGL	Qt OpenGL모듈에서 여러가지 식별자를 위한 이름공간
QGLColormap	QGLWidget들에 사용자정의색략도를 설치하는데 쓰인다.
QGLContext	OpenGL묘사상황을 은페한다.
QGLFormat	OpenGL묘사상황의 현시형식
QGLWidget	OpenGL그라픽스묘사용 창문부품
QIconSet	여러가지 형식과 크기의 그림기호모임
QImage	화소자료에 대한 직접호출을 가진 하드웨어독립픽스매프표시
QImageConsumer	QImageDecoder에 의하여 사용되는 추상화
QImageDecoder	유지되여있는 모든 화상형식의 증분식화상해신기
QImageFormat	특수화상형식의 증분식화상해신기
QImageFormatType	QImageFormat객체를 만드는 공장
QImageIO	화상의 적재와 보관용 파라메터
QMovie	동화상이나 정지화상의 적재상황을 신호하는 증분식적재
QPaintDevice	그리기할수 있는 객체들의 기초클라스
QPaintDeviceMetrics	그리기장치에 대한 정보
QPainter	창문부품들에 대한 저수준그리기를 한다.
QPalette	매개 창문부품상태를 위한 색그룹
QPen	QPainter가 직선과 륜곽선을 그리는 방법을 정의한다.
QPicture	QPainter지령을 기록하고 재연시하는 그리기장치
QPixmap	비직결화면, 화소에 기초한 그리기장치
QPixmapCache	픽스매프용의 응용프로그람대역캐쉬
QPNGImagePacker	잘 압축된 PNG동화상을 창조한다.
QPoint	평면안의 점을 정의한다.
QPointArray	점들의 배렬
Qprinter	인쇄기에 그리는 그리기장치
Qrect	평면안의 직사각형을 정의한다.
Qregion	그리기프로그람용의 복사칸(clip)령역
Qsize	2차원객체의 크기를 정의한다.
QWMatrix	자리표계의 2D변환

13) 배치관리

다음의 클라스들은 창문부품들의 자동기하학적(배치)관리를 제공한다.

QBoxLayout	자식창문부품들을 수평 혹은 수직으로 배치한다.
QButtonGroup	QButton창문부품들을 한 그룹으로 구성한다.
QGLayoutIterator	내부배치관리자반복자들의 추상기초클라스
QGrid	자식들의 단순한 기하학적관리
QGridLayout	창문부품을 격자형식으로 배치한다.
QGroupBox	제목있는 그룹칸프레임
QHBox	자식창문부품들의 수평기하학적관리

QHBoxLayout	창문부품들을 수평으로 배치한다.
QHButtonGroup	하나의 수평행을 가진 그룹으로 QButton창문부품들을 구성한다.
QHGroupBox	하나의 수평행을 가진 그룹으로 창문부품들을 구성한다.
QLayout	기하학적관리기들의 기초클라스
QLayoutItem	QLayout가 조작하는 추상적인 항목
QLayoutIterator	QLayoutItem에 대한 반복자
QSizePolicy	수평 및 수직크기조절방법을 서술하는 배치속성
QSpacerItem	배치관리자안의 빈 공간
QVBox	자식창문부품들의 수직기하학적관리
QVBoxLayout	창문부품들을 수직으로 배치한다.
QVButtonGroup	수직렬안의 QButton창문부품들을 조직한다.
QVGroupBox	수직렬안의 창문부품그룹을 조직한다.
QWidgetItem	한 창문부품을 표시하는 배치관리자항목

14) 암시적 및 명시적클라스

이 클라스들은 Qt에서 참고계수기와 공통자료에 최적화된 보통 무거운 클라스들이다. 유일하게 중요한 효과는 여기서 목록한 클라스들이 무거워보이지만 인수로서 효과적으로 넘길수 있는 클라스들이다.

QBitArray	비트배렬
QBitmap	흑백픽스매프(1 bit 깊이)
QBrush	QPainter로 그려질 도형들의 도색패턴을 정의한다.
QCString	전형적인 C의 0으로 완료되는 char배렬(char *)의 추상화
QCursor	임의의 모양의 마우스지적자
QDeepCopy	암시적 및 명시적 클라스들이 유일한 자료를 참고한다는것을 담보하는 형판클라스
QFont	본문그리기에 사용하는 서체
QFontInfo	서체에 대한 일반적 정보
QFontMetrics	서체크기정보
QIconSet	여러가지 형식과 크기의 그림기호모임
QImage	화소자료에 대한 직접호출을 가진 하드웨어독립픽스매프표시
QMap	사전을 제공하는 값에 기초한 형판클라스
QPair	한쌍의 원소들을 제공하는 값기초형판클라스
QPalette	매개 상태를 위한 색그룹
QPen	QPainter가 직선과 륜곽선을 그리는 방법을 정의한다.
QPicture	QPainter지령을 기록하고 재연시하는 그리기장치
QPixmap	비직결화면, 화소에 기초한 그리기장치
QPointArray	점들의 배렬
QRegExp	정규식들을 사용한 패턴대조
QString	유니코드본문과 전형적인 C의 '\0'완료문자배렬의 추상화
QStringList	문자렬목록
QValueList	목록을 제공하는 값기초형판클라스
QValueStack	탄창을 제공하는 값기초형판클라스
QValueVector	동적배렬을 제공하는 값기초형판클라스

15) 입출력과 망프로그람작성

다음의 클라스들은 외부장치, 프로세스, 파일로부터 입출력을 조종하며 파일과 등록부에 대하여 조작한다.

QBuffer	QByteArray에서 조작하는 I/O장치
QClipboard	창문체계오려둠판에 대한 호출
QDataStream	2진자료를 QIODevice에로의 계렬화
QDir	가동환경에 의존하지 않는 등록부구조와 그 내용호출
QDns	비동기DNS감시
QFile	파일에 조작하는 입출력장치
QFileInfo	체계에 의존하지 않는 파일정보
QFtp	FTP통신규약의 실현
QHostAddress	IP주소
QHttp	HTTP통신규약의 실현
QHttpHeader	HTTP용 머리부정보
QHttpRequestHeader	HTTP용 요구머리부정보
QHttpResponseHeader	HTTP용 응답머리부정보
QImageIO	화상의 적재와 보관용 파라메터
QIODevice	I/O장치들의 기초클라스
QLocalFs	국부파일체계에서 작업하는 QNetworkProtocol의 실현
QMacMime	열린표준형MIME를 Mac형식으로 변환한다.
QMimeSource	일정한 MIME형의 형식화된 자료를 제공하는 객체들의 추상화
QMimeSourceFactory	MIME형자료의 확장가능한 제공자
QNetworkOperation	망을 위한 일반조작
QNetworkProtocol	망용의 일반API
QProcess	외부크로그람들을 기동하고 그것들과 교제하는데 리용
QServerSocket	TCP에 기초한 봉사기
QSettings	가동환경에 의존하지 않는 영속적인 응용프로그람환경설정
QSignal	QObject를 계승하지 않는 클라스에 신호를 보낼 때 리용
QSignalMapper	식별가능한 송신자들로부터 오는 신호들을 묶는다.
QSocket	완충형 TCP련결
QSocketDevice	가동환경에 의존하지 않는 저수준소케트API
QSocketNotifier	소케트역호출용으로 제공
QTextIStream	입력스트림용 편의클라스
QTextOStream	출력스트림용 편의클라스
QTextStream	QIODevice를 사용하여 본문을 읽고 쓰는 기본함수들
QUrl	URL문법해석기와 그의 간단한 동작
QUrlInfo	URL에 대한 정보를 보관한다.
QUrlOperator	일반적인 URL조작
QWindowsMime	공개표준MIME를 Window오려둠판형식으로 변환한다.

16) 기본창문과 관련한 클라스들

다음의 클라스들은 기본창문자체와 차림표띠, 도구띠, 상태띠와 같이 현대적인 기본응용프로그람창문에 필요한 모든것을 제공한다.

QAction	차림표와 도구띠에 나타날 사용자대면부작용을 요약한다.
QActionGroup	작용들을 한 그룹에 묶는다.
QApplication	GUI 응용프로그람의 조종흐름과 기본설정을 관리한다.
QDockArea	QDockWindow들을 관리하고 배치한다.
QDockWindow	QDockArea안에서 류동할수 있거나 탁상우의 제일 웃준위 창문으로 될수 있는 창문부품
QEventLoop	사건대기렬을 관리한다.
QMainWindow	차림표띠 류동가능창문(례를 들면 도구띠), 상태띠가 있는 기본응용프로그람창문
QMenuBar	수평차림표띠
QPopupMenu	올리펼침차림표창문부품
QSessionManager	쎄숀관리기에 대한 호출
QSizeGrip	제일웃준위창문용 구석격자
QStatusBar	상태정보를 표시하는데 적합한 수평띠
QToolBar	도구단추와 같은 창문부품들을 포함하는 이동가능조종판
QWorkspace	MDI용과 같은 장식창문을 포함할수 있는 작업공간창문

17) 다매체클라스

다음의 클라스들은 그라픽스, 음성, 동화상 등의 기능을 제공한다.

QImageConsumer	QImageDecoder에 의해 사용되는 추상화
QImageDecoder	유지하고있는 모든 화상형식들의 증분식 화상해신기
QImageFormat	특정한 화상형식의 증분식화상해신기
QImageFormatType	QImageFormat객체들을 만드는 공장
QMovie	동화상 혹은 화상들의 증분식적재, 그 과정의 신호화
QSound	가동환경음성편의의 호출

18) 객체모형

다음의 클라스들은 Qt객체모형의 기초이다.

QGuardedPtr	QObject에로의 감시(guarded)지적자를 제공하는 형판클라스
QMetaObject	Qt객체에 대한 메타정보
QMetaProperty	속성에 대한 메타정보를 보관한다.
QObject	모든 Qt객체들의 기초클라스
QObjectCleanupHandler	여러 QObject들의 수명을 감시한다.
QVariant	가장 일반적인 Qt자료형들을 위한 공용체와 같이 동작한다.

19) 조직자

다음의 클라스들은 복잡한 응용프로그람 혹은 대화칸을 만드는데 쓰인다.

QButtonGroup	QButton창문부품들을 한 그룹으로 구성한다.
QGroupBox	제목있는 그룹칸프레임

QHBox	자식창문부품들의 수평기하학적관리
QHButtonGroup	하나의 수평행을 가진 그룹으로 QButton창문부품들을 구성한다.
QHGroupBox	하나의 수평행을 가진 그룹으로 창문부품들을 구성한다.
QSplitter	분할창문부품을 실현한다.
QTabWidget	타브형 창문부품들의 탄창
QVBox	자식창문부품들의 수직기하학적관리
QVButtonGroup	수직렬안의 QButton창문부품들을 조직한다.
QVGroupBox	수직렬안의 창문부품그룹을 조직한다.
QWidgetStack	제일 웃끝의 창문부품만 볼수 있는 창문부품탄창
QWizard	위자드대화칸용 틀거리
QWorkspace	MDI용과 같은 장식창문을 포함할수 있는 작업공간창문

20) 플라그인

다음의 클라스들은 공유서고(실례로 .so와 DLL파일들)과 Qt플라그인들을 론한다.

QGfxDriverPlugin	Qt/Embedded그라픽스구동프로그람플라그인용 추상기초클라스
QImageFormatPlugin	사용자정의화상형식플라그인용 추상기초클라스
QKbdDriverPlugin	Qt/Embedded건반구동프로그람플라그인용 추상기초클라스
QLibrary	공유서고를 조종하기 위한 래퍼
QMouseDriverPlugin	Qt/Embedded마우스구동프로그람플라그인용 추상기초클라스
QSqlDriverPlugin	사용자정의QSqlDriver플라그인용 추상기초클라스
QStylePlugin	사용자정의QStyle플라그인용 추상기초클라스
QTextCodecPlugin	사용자정의QTextCodec플라그인용 추상기초클라스
QWidgetPlugin	사용자정의QWidget플라그인용 추상기초클라스

21) 대화칸클라스

다음의 클라스들은 더 간단한 창문부품 즉 일반적으로 대화칸으로 구성되는 복합창문부품이다.

QColorDialog	색을 지정하기 위한 대화칸창문부품
QDialog	대화창문의 기초클라스
QErrorMessage	오유통보현시대화칸
QFileDialog	사용자가 파일이나 등록부를 선택하게 하는 대화칸
QFontDialog	서체를 선택하는 대화칸창문부품
QInputDialog	사용자로부터 단일값을 얻기 위한 단순하고 편리한 대화칸
QMessageBox	간단한 통보문, 그림기호, 몇개의 단추들을 가진 이행금지대화칸
QProgressDialog	느린 조작의 진척상황 제공
QTabDialog	타브형 창문부품들의 탄창
QWizard	위자드대화칸용 틀거리

22) Qt형판서고클라스

Qt Template Library (QTL)은 객체용기를 제공하는 형판들의 모임이다.

QMap	사전을 제공하는 값에 기초한 형판클라스
QMapConstIterator	QMap의 상수반복자
QMapIterator	QMap의 반복자
QPair	한쌍의 원소들을 제공하는 값기초형판클라스
QValueList	목록을 제공하는 값기초형판클라스
QValueListConstIterator	QValueList의 상수반복자
QValueListIterator	QValueList의 반복자
QValueStack	탄창을 제공하는 값기초형판클라스
QValueVector	동적배렬을 제공하는 값기초형판클라스

23) 본문관련클라스

다음의 클라스들은 본문처리와 관련된다.

QChar	간략유니코드문자
QCharRef	QString용 방조클라스
QConstString	상수유니코드자료를 사용하는 문자렬객체
QCString	전형적인 C의 0으로 완료되는 char배렬(char *)의 추상화
QLabel	본문 혹은 화상현시
QLocale	수들과 여러가지 언어로 된 그것들의 문자렬표현사이의 변환
QSimpleRichText	현시할수 있는 자그마한 리치본문부분
QString	유니코드본문과 전형적인 C의 '\0'완료문자배렬의 추상화
QStringList	문자렬목록
QStrList	char*의 2중련결목록
QStyleSheet	리치본문묘사용의 형식들의 집합과 타그생성기
QStyleSheetItem	일련의 본문형식들의 은페
QSyntaxHighlighter	QTextEdit문법강조표시기를 실현하는 기초클라스
QTextBrowser	초본문항행이 가능한 리치본문열람기
QTextEdit	강력한 단일페지리치본문편집기
QTextIStream	입력스트림용 편의클라스
QTextOStream	출력스트림용 편의클라스
QTextStream	QIODevice를 사용하여 본문을 읽고 쓰는 기본함수들

24) 스레드화통보

다음의 클라스들은 스레드화된 응용프로그람들과 관련이 있다.

QMutex	스레드들사이의 호출계렬화
QMutexLocker	QMutexe의 잠그기와 해제를 간단화한다.
QSemaphore	건전한 옹근쎄마훠
QThread	가동환경에 의존하지 않는 스레드
QThreadStorage	스레드별 자료기억
QWaitCondition	스레드들사이의 조건들에 대한 대기/동작(waking)을 허용한다.

25) 창문부품모양과 형식

다음의 클라스들은 응용프로그람의 모양과 형식을 전용화하는데 쓰인다.

QBoxLayout	자식창문부품들을 수평 혹은 수직으로 배치한다.
QButtonGroup	그룹안의 QButton창문부품들을 조직한다.
QCDEStyle	CDE형식
QColor	RGB 혹은 HSV값에 기초하는 색
QColorGroup	창문부품색의 그룹
QCommonStyle	GUI 의 일반적형식을 은페한다.
QCursor	임의의 모양의 마우스지적자
QFont	본문그리기에 사용하는 서체
QGLayoutIterator	내부배치관리자반복자들의 추상기초클라스
QGrid	자식들의 단순한 기하학적관리
QGridLayout	창문부품을 격자로 배치한다.
QGroupBox	제목있는 그룹칸프레임
QHBox	자식창문부품들의 수평기하학적관리
QHBoxLayout	창문부품들을 수평으로 배치한다.
QHButtonGroup	하나의 수평행을 가진 그룹으로 QButton창문부품들을 구성한다.
QHGroupBox	하나의 수평행을 가진 그룹으로 창문부품들을 구성한다.
QLayout	기하학적관리기들의 기초클라스
QLayoutItem	QLayout가 조작하는 추상적인 항목
QLayoutIterator	QLayoutItem에 대한 반복자
QMacStyle	모양관리기(Appearance Manager)형식을 실현한다.
QMotifPlusStyle	복잡한 Motif형식
QMotifStyle	Motif형식
QPalette	매개 상태를 위한 색그룹
QPlatinumStyle	Mac/Platinum형식
QSGIStyle	SGI/Irix형식
QSizeGrip	제일웃준위창문용 구석격자
QSizePolicy	수평 및 수직크기조절방법을 서술하는 배치관리자속성
QSpacerItem	배치관리자에서 빈 공간
QStyle	GUI의 형식
QStyleOption	QStyle함수용 선택파라메터
QVBox	자식창문부품들의 수직기하학적관리
QVBoxLayout	창문부품들을 수직으로 배치한다.
QVButtonGroup	수직렬안의 QButton창문부품들을 조직한다.
QVGroupBox	수직렬안의 창문부품그룹을 조직한다.
QWidgetItem	한 창문부품을 표시하는 배치관리자항목
QWindowsStyle	Microsoft Windows형식

26) XML

다음의 클라스들은 XML사용자들과 관련이 있다.

QDomAttr	QDomElement 의 한개 속성을 표시한다.
QDomCDATASection	XML CDATA 절을 표시한다.

QDomCharacterData	DOM 안의 일반문자렬의 표시한다.
QDomComment	XML 설명문을 표시한다.
QDomDocument	XML 문서를 표시한다.
QDomDocumentFragment	보통 완전한 QDomDocument 이 아닌 QDomNode 들의 나무
QDomDocumentType	문서나무에서 DTD 의 표현
QDomElement	DOM 나무의 한 요소를 표시한다.
QDomEntity	XML 실체를 표시한다.
QDomEntityReference	XML 실체참고를 표시한다.
QDomImplementation	DOM 실현의 특성에 대한 정보
QDomNamedNodeMap	이름에 의해 호출될수 있는 마디집합
QDomNode	DOM 나무의 모든 마디들의 기초클라스
QDomNodeList	QDomNode 객체들의 목록
QDomNotation	XML 표기법을 표시한다.
QDomProcessingInstruction	XML 처리지령을 표시한다.
QDomText	문법해석된 XML 문서의 본문자료를 표시한다.
QXmlAttributes	XML 속성
QXmlContentHandler	XML 자료의 론리적내용을 알리기 위한 대면부
QXmlDeclHandler	XML 자료의 선언내용을 알리기 위한 대면부
QXmlDefaultHandler	모든 XML 처리함수클라스들의 기정실현
QXmlDTDHandler	XML 자료의 DTD 내용을 알리기 위한 대면부
QXmlEntityResolver	XML 자료에 포함된 외부실체를 해결하기 위한 대면부
QXmlErrorHandler	XML 자료로 오유를 알리기 위한 대면부
QXmlInputSource	QXmlReader 파생클라스용 입력자료
QXmlLexicalHandler	XML 자료의 어휘적내용을 알리기 위한 대면부
QXmlLocator	파일안에서 문법해석위치에 대한 정보를 가지는 XML 처리함수 클라스
QXmlNamespaceSupport	이름공간유지를 포함하려고 하는 XML 읽기기구용 방조클라스
QXmlParseException	QXmlErrorHandler 대면부에서 오유통보에 사용한다.
QXmlReader	XML 읽기기구(문법해석기)의 대면부
QXmlSimpleReader	간단한 XML 읽기기구(문법해석기)의 실현

27) 기타 클라스

다음의 클라스들은 다른 범주에 속하지 않는 쓸모있는 클라스들이다.

QAccel	건반지름건과 지름건을 조종한다.
QAccessible	호출가능성과 관련한 렬거형 및 정적함수들
QAccessibleInterface	호출가능한 객체들에 대한 정보를 발생하는 대면부를 정의한다.
QAccessibleObject	QObject 용 QAccessibleInterface 의 부분들을 실현한다.
QCustomMenuItem	올리펼침차림표에서 사용자정의차림표항목들을 위한 추상기초클라스

QDoubleValidator	류동소수점수의 범위검사
QErrorMessage	오유통보현시대화칸
QFileIconProvider	사용하려는 QFileDialog 용 그림기호
QFilePreview	QFileDialog 에서 파일미리보기
QFocusData	초점련쇄안의 창문부품목록을 관리한다.
QIntValidator	문자렬이 지정된 범위안의 유효옹근수를 포함한다는것을 담보하는 유효성검사기
QKeySequence	지름건으로 사용되는 건들의 렬을 은폐한다.
QMacMime	열린표준형 MIME 를 Mac 형식으로 변환한다.
QMenuData	QMenuBar 와 QPopupMenu 의 기초클라스
QMimeSource	일정한 MIME 형의 형식화된 자료를 제공하는 객체들의 추상화
QProcess	외부크로그람들을 기동하고 그것들과 교제하는데 리용
QRangeControl	범위안의 옹근수값
QRegExp	정규식들을 사용한 패턴대조
QRegExpValidator	정규식에 대하여 문자렬을 검사하는데 리용
QSettings	가동환경에 의존하지 않는 영속적인 응용프로그람환경설정
QSignal	QObject 를 계승하지 않는 클라스에 신호를 보낼 때 리용
QSplashScreen	응용프로그람을 기동할 때 보여주는 스플라쉬화면
Qt	대역이여야 하는 여러가지 식별자들을 위한 이름공간
QUrl	URL 문법해석기와 그의 간단한 동작
QUrlInfo	URL 에 대한 정보를 보관한다.
QUrlOperator	일반적인 URL 조작
QValidator	입력본문의 확증
QVariant	가장 일반적인 Qt 자료형들을 위한 공용체와 같이 동작한다.
QWindowsMime	공개표준 MIME 를 Window 오려둠판형식으로 변환한다.

3. 집합클라스

집합클라스는 많은 항목들을 자료구조에 보관하는 용기로서 항목의 삽입과 삭제, 검색과 같은 집합의 내용에 대한 여러가지 조작을 제공한다.

Qt에는 여러개의 값기초집합클라스들과 지적자기초집합클라스들이 있다. 지적자기초집합클라스는 항목에로의지적자들과 작업하며 값기초클라스들은 그 항목들의 사본들을 복사한다. 값기초집합은 STL용기클라스들과 비슷하며 STL알고리듬 및 용기들과 함께 쓰일수도 있다.

값에 기초한 집합은 다음과 같다.

QValueList, 값기초목록.

QValueVector, 값기초벡토르.

QValueStack, 값기초탄창.

QMap, 값기초사전(련상배렬).

지적자에 기초한 집합은 다음과 같다.

QCache와 QIntCache, 적어도 최근에 사용된(LRU, least recently used) 캐쉬.

QDict, QIntDict 그리고 QPtrDict사전.

QPtrList, 2중련결목록.

QPtrQueue, 처음 넣은것을 처음에 꺼내는 (FIFO, first in, first out) 대기렬.

QPtrStack, 마지막으로 넣은것을 처음에 꺼내는 (LIFO, last in, first out) 탄창.
QPtrVector, 벡토르.
QMemArray는 례외로서 지적자나 값에 기초하지 않고 기억기기초이다. 단순자료형을 리용할 때의 최대효과성은 보통 배렬에 사용되며 배렬원소들의 복사와 비교에 비트연산을 리용한다.
이 클라스들중 일부는 대응하는 반복자를 가지고있다. 반복자는 집합안의 항목들을 항행하기 위한 클라스이다.
QCacheIterator와 QIntCacheIterator
QDictIterator와 QIntDictIterator, QPtrDictIterator
QPtrListIterator
QValueListIterator와 QValueListConstIterator
QMapIterator와 QMapConstIterator
값기초집합과 그것에 대하여 조작하는 알고리듬은 Qt형판서고에서 분류하였다.

1) 지적자기초용기의 구성방식

지적자기초용기를 위한 내부기초클라스가 4개(QGCache와 QGDict, QGList, QGVector) 있으며 그것들은 void지적자에 대하여 조작한다. 빈약한 형판층은 항목들과 void지적자들사이의 강제변환에 의하여 실제의 집합을 실현한다.
이 전략은 Qt형판들의 성능을 떨구지 않고 공간에 대하여 아주 효과적이도록 한다. (즉 이 형판들중 하나의 실례를 만들 때 기초클라스에로의 inline호출만 추가한다.)

2) QPtrList실례

이 실례는 목록에 Employee항목들을 보관하고 그것들을 반대순서로 출력하는 방법을 보여준다.

```
#include <qptrlist.h>
#include <qstring.h>
#include <stdio.h>

class Employee
{
public:
 Employee( const char *name, int salary ) { n=name; s=salary; }
 const char *name()    const       { return n; }
 int    salary() const       { return s; }
private:
 QString  n;
 int    s;
};

int main()
{
 QPtrList<Employee> list;  // list of pointers to Employee
 list.setAutoDelete( TRUE );  // delete items when they are removed

 list.append( new Employee("Bill", 50000) );
 list.append( new Employee("Steve",80000) );
 list.append( new Employee("Ron",  60000) );
```

```
  QPtrListIterator<Employee> it(list); // iterator for employee list
  for ( it.toLast(); it.current(); --it) ) {
   Employee *emp = it.current();
   printf( "%s earns %d\n", emp->name(), emp->salary() );
  }

  return 0;
}
```
프로그람출력:

```
 Ron earns 60000
 Steve earns 80000
 Bill earns 50000
```

3) 집합항목들의 관리

모든 지적자기초집합은 QPtrCollection기초클라스를 계승한다. 이 클라스는 집합안의 항목수와 삭제전략에 대해서만 알고있다.

기정으로 집합안의 항목들은 집합에서 삭제될 때 삭제되지 않는다. QPtrCollection::setAutoDelete()함수는 삭제전략을 지정한다. 이 실례에서는 자동삭제를 허용하여 항목들이 목록에서 삭제될 때 항목자체를 삭제하게 한다.

집합에 항목을 삽입할 때 지적자만 복사되고 항목자체는 복사되지 않는다. 이것을 얕은 복사(shallow copy)라고 한다. 항목이 삽입될 때 집합에 항목자료모두를 복사하게 할수 있다. (이것을 깊은 복사(deep copy)라고 한다.) 항목을 삽입하는 집합함수들은 삽입하려는 항목에 대하여 가상함수 QPtrCollection::newItem()를 호출한다. 자기의 집합에서 깊은 복사를 유지하려고 한다면 집합을 계승하여 그것을 재정의한다.

목록으로부터 항목을 삭제할 때 가상함수 QPtrCollection::deleteItem()가 호출된다. 모든 집합클라스들의 기정실현에서는 자동삭제가 허용되여있으면 항목이 삭제된다.

4) 사용법

QPtrList<type>와 같은 지적자기초클라스는 *type*객체의 지적자집합을 정의한다. 지적자(*)는 무조건적이다.

여기서는 QPtrList를 론하지만 같은 수법을 모든 지적자기초집합클라스들과 모든 집합클라스반복자들에 적용한다.

형판실례의 창조:

```
 QPtrList<Employee> list;   // 목록사용
```

실례에서 항목의 클라스 혹은 형 Employee는 목록정의에 앞서 정의되여야 한다.

```
 // Employee가 정의되지 않았으므로 작업하지 않는다.
 class Employee;
 QPtrList<Employee> list;

 // Employee가 리용되기전에 정의되므로 다음 코드는 작업한다.
 class Employee {
  ...
 };
 QPtrList<Employee> list;
```

5) 반복자

QPtrList는 목록을 항행하는 성원함수들을 가지지만 흔히 반복자를 사용하는것이 더 좋다. QPtrListIterator는 아주 안전하고 동시에 수정되고있는 목록들을 항행할수 있다. 여러 반복자들이 같은 집합에서 독립적으로 작업할수 있다.

QPtrList은 현재 거기에 조작하는 모든 반복자들의 내부목록을 가지고있다. 목록항목이 삭제될 때 목록은 그에 따라 모든 반복자들을 갱신한다.

QDict와 QCache집합에는 항행함수가 없다. 이 집합을 항행하려면 QDictIterator 혹은 QCacheIterator를 사용해야 한다.

6) 미리 정의된 집합

Qt의 미리 정의된 집합클라스는 다음과 같다.

- 문자렬목록: QStrList과 QStrIList (qstrlist.h), QStringList (qstringlist.h)
- 문자렬벡토르: QStrVec와 QStrIVec (qstrvec.h); 이것들은 낡은것이다.

대부분의 경우에 절대공유된 QString유니코드문자렬의 값목록인 QStringList를 선택할수 있다. QPtrStrList과 QPtrStrIList는 문자렬자체가 아니라 char지적자만 보관한다.

7) 지적자기초집합클라스와 관련한 반복자클라스들의 목록

QAsciiCache	char*건에 기초한 캐쉬를 제공하는 형판클라스
QAsciiCacheIterator	QAsciiCache 집합용 반복자
QAsciiDict	char*건에 기초한 사전을 제공하는 형판클라스
QAsciiDictIterator	QAsciiDict 집합용 반복자
QBitArray	비트배렬
QBitVal	QBitArray 와 함께 쓰이는 내부클라스
QBuffer	QByteArray 에서 조작하는 I/O 장치
QByteArray	바이트배렬
QCache	QString 건에 기초한 캐쉬를 제공하는 형판클라스
QCacheIterator	QCache 집합용 반복자
QCString	C 의 령마감 char 배렬(char *)의 추상화
QDict	QString 건에 기초한 사전을 제공하는 형판클라스
QDictIterator	QDict 집합의 반복자
QIntCache	긴열쇠에 기초하는 캐쉬를 제공하는 형판클라스
QIntCacheIterator	QIntCache 집합의 반복자
QIntDict	긴열쇠에 기초하는 등록부를 제공하는 형판클라스
QIntDictIterator	QIntDict 집합의 반복자
QObjectList	QObject 들의 QPtrList
QObjectListIterator	QObjectList 용 반복자
QPtrCollection	대다수 지적자에 기초한 Qt 집합의 기초클라스
QPtrDict	void*건에 기초하는 사전을 제공하는 형판클라스
QPtrDictIterator	QPtrDict 집합용 반복자
QPtrList	목록을 제공하는 형판클라스
QPtrListIterator	QPtrList 집합용 반복자
QPtrQueue	대기렬을 제공하는 형판클라스
QStrIList	대소문자식별비교기능을 가진 char*의 2 중련결목록
QStrList	char*의 2 중련결목록

부록 2. 수 2-9999의 씨수표

2	3	5	7	11	13	17	19	23	29
31	37	41	43	47	53	59	61	67	71
73	79	83	89	97	101	103	107	109	113
127	131	137	139	149	151	157	163	167	173
179	181	191	193	197	199	211	223	227	229
233	239	241	251	257	263	269	271	277	281
283	293	307	311	313	317	331	337	347	349
353	359	367	373	379	383	389	397	401	409
419	421	431	433	439	443	449	457	461	463
467	479	487	491	499	503	509	521	523	541
547	557	563	569	571	577	587	593	599	601
607	613	617	619	631	641	643	647	653	659
661	673	677	683	691	701	709	719	727	733
739	743	751	757	761	769	773	787	797	809
811	821	823	827	829	839	853	857	859	863
877	881	883	887	907	911	919	929	937	941
947	953	967	971	977	983	991	997	1009	1013
1019	1021	1031	1033	1039	1049	1051	1061	1063	1069
1087	1091	1093	1097	1103	1109	1117	1123	1129	1151
1153	1163	1171	1181	1187	1193	1201	1213	1217	1223
1229	1231	1237	1249	1259	1277	1279	1283	1289	1291
1297	1301	1303	1307	1319	1321	1327	1361	1367	1373
1381	1399	1409	1423	1427	1429	1433	1439	1447	1451
1453	1459	1471	1481	1483	1487	1489	1493	1499	1511
1523	1531	1543	1549	1553	1559	1567	1571	1579	1583
1597	1601	1607	1609	1613	1619	1621	1627	1637	1657
1663	1667	1669	1693	1697	1699	1709	1721	1723	1733
1741	1747	1753	1759	1777	1783	1787	1789	1801	1811
1823	1831	1847	1861	1867	1871	1873	1877	1879	1889
1901	1907	1913	1931	1933	1949	1951	1973	1979	1987
1993	1997	1999	2003	2011	2017	2027	2029	2039	2053
2063	2069	2081	2083	2087	2089	2099	2111	2113	2129
2131	2137	2141	2143	2153	2161	2179	2203	2207	2213
2221	2237	2239	2243	2251	2267	2269	2273	2281	2287
2293	2297	2309	2311	2333	2339	2341	2347	2351	2357
2371	2377	2381	2383	2389	2393	2399	2411	2417	2423
2437	2441	2447	2459	2467	2473	2477	2503	2521	2531
2539	2543	2549	2551	2557	2579	2591	2593	2609	2617
2621	2633	2647	2657	2659	2663	2671	2677	2683	2687
2689	2693	2699	2707	2711	2713	2719	2729	2731	2741
2749	2753	2767	2777	2789	2791	2797	2801	2803	2819
2833	2837	2843	2851	2857	2861	2879	2887	2897	2903
2909	2917	2927	2939	2953	2957	2963	2969	2971	2999
3001	3011	3019	3023	3037	3041	3049	3061	3067	3079
3083	3089	3109	3119	3121	3137	3163	3167	3169	3181
3187	3191	3203	3209	3217	3221	3229	3251	3253	3257
3259	3271	3299	3301	3307	3313	3319	3323	3329	3331
3343	3347	3359	3361	3371	3373	3389	3391	3407	3413
3433	3449	3457	3461	3463	3467	3469	3491	3499	3511

3517	3527	3529	3533	3539	3541	3547	3557	3559	3571
3581	3583	3593	3607	3613	3617	3623	3631	3637	3643
3659	3671	3673	3677	3691	3697	3701	3709	3719	3727
3733	3739	3761	3767	3769	3779	3793	3797	3803	3821
3823	3833	3847	3851	3853	3863	3877	3881	3889	3907
3911	3917	3919	3923	3929	3931	3943	3947	3967	3989
4001	4003	4007	4013	4019	4021	4027	4049	4051	4057
4073	4079	4091	4093	4099	4111	4127	4129	4133	4139
4153	4157	4159	4177	4201	4211	4217	4219	4229	4231
4241	4243	4253	4259	4261	4271	4273	4283	4289	4297
4327	4337	4339	4349	4357	4363	4373	4391	4397	4409
4421	4423	4441	4447	4451	4457	4463	4481	4483	4493
4507	4513	4517	4519	4523	4547	4549	4561	4567	4583
4591	4597	4603	4621	4637	4639	4643	4649	4651	4657
4663	4673	4679	4691	4703	4721	4723	4729	4733	4751
4759	4783	4787	4789	4793	4799	4801	4813	4817	4831
4861	4871	4877	4889	4903	4909	4919	4931	4933	4937
4943	4951	4957	4967	4969	4973	4987	4993	4999	5003
5009	5011	5021	5023	5039	5051	5059	5077	5081	5087
5099	5101	5107	5113	5119	5147	5153	5167	5171	5179
5189	5197	5209	5227	5231	5233	5237	5261	5273	5279
5281	5297	5303	5309	5323	5333	5347	5351	5381	5387
5393	5399	5407	5413	5417	5419	5431	5437	5441	5443
5449	5471	5477	5479	5483	5501	5503	5507	5519	5521
5527	5531	5557	5563	5569	5573	5581	5591	5623	5639
5641	5647	5651	5653	5657	5659	5669	5683	5689	5693
5701	5711	5717	5737	5741	5743	5749	5779	5783	5791
5801	5807	5813	5821	5827	5839	5843	5849	5851	5857
5861	5867	5869	5879	5881	5897	5903	5923	5927	5939
5953	5981	5987	6007	6011	6029	6037	6043	6047	6053
6067	6073	6079	6089	6091	6101	6113	6121	6131	6133
6143	6151	6163	6173	6197	6199	6203	6211	6217	6221
6229	6247	6257	6263	6269	6271	6277	6287	6299	6301
6311	6317	6323	6329	6337	6343	6353	6359	6361	6367
6373	6379	6389	6397	6421	6427	6449	6451	6469	6473
6481	6491	6521	6529	6547	6551	6553	6563	6569	6571
6577	6581	6599	6607	6619	6637	6653	6659	6661	6673
6679	6689	6691	6701	6703	6709	6719	6733	6737	6761
6763	6779	6781	6791	6793	6803	6823	6827	6829	6833
6841	6857	6863	6869	6871	6883	6899	6907	6911	6917
6947	6949	6959	6961	6967	6971	6977	6983	6991	6997
7001	7013	7019	7027	7039	7043	7057	7069	7079	7103
7109	7121	7127	7129	7151	7159	7177	7187	7193	7207
7211	7213	7219	7229	7237	7243	7247	7253	7283	7297
7307	7309	7321	7331	7333	7349	7351	7369	7393	7411
7417	7433	7451	7457	7459	7477	7481	7487	7489	7499
7507	7517	7523	7529	7537	7541	7547	7549	7559	7561
7573	7577	7583	7589	7591	7603	7607	7621	7639	7643
7649	7669	7673	7681	7687	7691	7699	7703	7717	7723
7727	7741	7753	7757	7759	7789	7793	7817	7823	7829
7841	7853	7867	7873	7877	7879	7883	7901	7907	7919
7927	7933	7937	7949	7951	7963	7993	8009	8011	8017

8039 8053 8059 8069 8081 8087 8089 8093 8101 8111
8117 8123 8147 8161 8167 8171 8179 8191 8209 8219
8221 8231 8233 8237 8243 8263 8269 8273 8287 8291
8293 8297 8311 8317 8329 8353 8363 8369 8377 8387
8389 8419 8423 8429 8431 8443 8447 8461 8467 8501
8513 8521 8527 8537 8539 8543 8563 8573 8581 8597
8599 8609 8623 8627 8629 8641 8647 8663 8669 8677
8681 8689 8693 8699 8707 8713 8719 8731 8737 8741
8747 8753 8761 8779 8783 8803 8807 8819 8821 8831
8837 8839 8849 8861 8863 8867 8887 8893 8923 8929
8933 8941 8951 8963 8969 8971 8999 9001 9007 9011
9013 9029 9041 9043 9049 9059 9067 9091 9103 9109
9127 9133 9137 9151 9157 9161 9173 9181 9187 9199
9203 9209 9221 9227 9239 9241 9257 9277 9281 9283
9293 9311 9319 9323 9337 9341 9343 9349 9371 9377
9391 9397 9403 9413 9419 9421 9431 9433 9437 9439
9461 9463 9467 9473 9479 9491 9497 9511 9521 9533
9539 9547 9551 9587 9601 9613 9619 9623 9629 9631
9643 9649 9661 9677 9679 9689 9697 9719 9721 9733
9739 9743 9749 9767 9769 9781 9787 9791 9803 9811
9817 9829 9833 9839 9851 9857 9859 9871 9883 9887
9901 9907 9923 9929 9931 9941 9949 9967 9973

참고문헌

1. Trolltech 《Qt 3.3.6 Reference Documentation》 Trolltech, 2005.

2. C. J. Date 《An Introduction to Database Systems (7th ed.)》 ISBN, 0201385902.

이 책은 콤퓨터를 전공으로 하는 교원, 연구사, 대학생들을 위한 참고서이다.

Qt일반지식

집 필	한영철,홍이철,문홍남	**심 사**	박사 부교수 김순희
편 집	차현옥	**교 정**	서금석
장 정	서경애	**콤퓨터편성**	여은정
낸 곳	교육위원회 교육정보쎈터	**인쇄소**	
인 쇄		**발 행**	

교-09-1663　　부　　값　　원